Parametrischer Polymorphismus, Überladungen und Konversionen

Vom Fachbereich Informatik der
Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
genehmigte Dissertation
von

Dipl.-Inform. Stefan Kaes

aus
Frankfurt am Main

Referent: Prof. Dr. Thomas Kühne
Koreferent: Prof. Dr. Wolfgang Henhapl

Tag der Einreichung: 28.1.2005
Tag der mündlichen Prüfung: 3.3.2005

Darmstadt 2005
D 17
Darmstädter Dissertation

Zusammenfassung

Typsysteme auf Basis des von R. Milner entwickelten Konzepts des *parametrischen Polymorhpismus* zeichnen sich dadurch aus, daß jedes wohltypisierte Programm eine eindeutige denotationale Semantik besitzt, die Typfehler zur Laufzeit verhindert. Darüber hinaus kann jedem typisierbaren Programm ein allgemeinster Typ zugeordnet werden, der alle möglichen Instanztypen durch einen einzigen Typausdruck über unsortierten Typvariablen darstellt, und dieser Typausdruck kann auf Basis eines Unifikationsalgorithmus für Typausdrücke berechnet werden. Dabei sind Typangaben für vom Programmierer eingeführte Bezeichner optional.

Überladene Operatoren bzw. Konstanten werden in traditionellen und auch in moderneren objekt-orientierten Sprachen als rein syntaktische Erleichterungen behandelt: ein Operator darf eine endliche Menge verschiedener Typen annehmen und zur Übersetzungszeit muß für sämtliche Vorkommen eines überladenen Operators bestimmt werden, welche Überladungsinstanz vom Programmierer an der entsprechenden Programmstelle intendiert war. Diesen Prozeß bezeichnet man üblicherweise als *Überladungsauflösung*.

Fügt man einem parametrisch polymorphen Typsystem Überladungen auf der Grundlage dieses Ansatzes hinzu, dann erhält man einen NP-vollständigen Typinferenzalgorithmus. Darüber hinaus kann ein solches Typsystem zwar die Verwendung überladener Operatoren ermöglichen, aber es erlaubt nicht deren Abstraktion in der Definition neuer Funktionen, da ein Ausdruck mehr als einen Typ besitzen kann.

In der vorliegenden Arbeit wird das Konzept der *parametrischen Überladungen* vorgestellt, das diese Nachteile vermeidet und dabei sämtliche Vorteile des parametrischen Polymorphismus beibehält: jeder typisierbare Ausdruck besitzt einen allgemeinsten Typ, typisierbare Programme führen nie zu Laufzeitfehlern, die Komplexität des Inferenzalgorithmus ist polynomial zeitbeschränkt und überladene Operatoren können in Funktionsdefinitionen abstrahiert werden.

Darauf aufbauend untersuchen wir Typinferenzsysteme für Sprachen, die zusätzlich zu Überladungen auch noch *implizite Konversionen* unterstützen. Dazu definieren wir ein *generisches* Typinferenzsystem für *eingeschränkte Typen* und zeigen, daß auch in einem solchen System allgemeinste Typen existieren und Typisierbarkeit

für bestimmte Restriktionsklassen entscheidbar bleibt. Abschließend zeigen wir, daß parametrische Überladungen mit *regulären* Typen kombiniert werden können.

Abstract

Type systems based on R. Milner's concept of *parametric polymorphism* are recognized by the following facts: every well typed program has a uniquely defined denotational semantics, which ensures that the evaluation of a program will never produce a runtime type error. Additionally, every well typed program has a *principal type* such that every other type is an instance of the principal type. The principal type is a type expression over unsorted variables and can be computed by a type inference algorithm based on Robinson's unification algorithm.

In traditional programming languages, overloaded operators and constants are treated as pure syntactic sugar: an overloaded operator can have a finite number of unrelated types and the type analysis component must determine for each occurrence of an overloaded operator, which of its overloaded instances was intended by the programmer. This process is usually called *overloading resolution*.

Adding overloaded operators based on this concept to a parametrically polymorphic language has two drawbacks: since each programmer declared identifier is reqired to have a simple type, the type inference algorithm becomes NP-complete and overloaded operators cannot be abstracted over in user defined functions.

In this thesis we define the concept of *parametric overloading*, which avoids these problems and keeps all nice properties of parametric polymorphism: principality of types, absence of runtime errors for well typed programs, polynomially time bounded type inference algorithm and full abstraction over overloaded operators.

Building on these results, we investigate a type system for a language which additionally supports *implicit coercions*. We define a *generic* inference system based on the concept of *constrained types*, show that principal types exist for this system and typability remains decidable for a special class of constraints. Finally we show that parametric overloading can be combined with *regular types*.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

Zwischen 1977 und 1987 wurde eine große Zahl funktionaler Programmiersprachen entwickelt, deren Typsystem auf dem von Robin Milner entworfenen Konzept des parametrischen Polymorphismus [Mil78] basiert. Die bekannteren dieser Sprachen sind ML, bzw. Standard ML [Mil84], HOPE [BMS80] und Miranda [Tur85]. Dieser Erfolg des Konzepts beruht nach Ansicht des Autors auf der Kombination einer Reihe von Tatsachen:

Sicherheit: Die Typkorrektheit von Programmen wird vollständig zur Übersetzungszeit überprüft und ermöglicht damit die Entdeckung einer großen Anzahl von Programmierfehlern zu Beginn des Programmentwicklungsprozesses. Typisierbare Programme haben die Eigenschaft, daß zur Laufzeit keine Typfehler auftreten können.

Flexibilität: Parametrischer Polymorphismus erlaubt dem Programmierer die Verwendung von Funktionen für Argumente verschiedenen Typs, unter der Voraussetzung, daß die Bedeutung der Funktion unabhängig von den Typen der Argumente ist. Damit wird gleichzeitig die Entwicklung wiederverwendbarer Software unterstützt.

Bequemlichkeit: Aufgrund der Inferierbarkeit der Typen aller deklarierten Bezeichner, wird der Programmierer bzw. Spezifizierer nicht mit unnötigen, oft trivialen Typangaben belastet.[1]

Effizienz: Da alle Typüberprüfungen zur Übersetzungszeit durchgeführt werden, können teure Typüberprüfungen zur Ausführungszeit vermieden werden, was als Nebeneffekt zu einer beträchtlichen Effizienzsteigerung funktionaler Sprachimplementierungen geführt hat.

Unterzieht man das Konzept des parametrischen Polymorphismus einer genauen Prü-

[1] Dies soll natürlich nicht heißen, daß der Autor dafür plädiert, daß Typangaben prinzipiell weggelassen werden sollten. Zu Dokumentationszwecken ist es oft sinnvoll, zumindest für global definierte Bezeichner die Typen explizit anzugeben.

fung, so erkennt man jedoch, daß sowohl die Flexibilität als auch die Sicherheit zur Beschreibung üblicher Programmiersprachen im Allgemeinen nicht ausreichen. Wir betrachten dazu als Beispiel einige der üblichen, vordefinierten, relationalen bzw. arithmetischen Operatoren in einer Sprache, die sowohl ganze als auch reelle Zahlen sowie homogene Listen und Funktionen als Datentypen enthält.

Typisches Beispiel für eine parametrisch polymorphe Funktion ist die Funktion *len* zur Berechnung der Länge einer Liste von Werten beliebigen, aber gleichen Typs:

$$len\ l = \text{if } null\ l \text{ then } 0 \text{ else } 1 + len(tl\ l)$$

In einem parametrisch polymorphen Typsystem wird *len* der Typ $\forall t.list(t) \rightarrow int$ zugeordnet, wobei t für einen beliebigen Typ steht. Den polymorphen Typausdruck $\forall t.list(t) \rightarrow int$ kann man als Beschreibung einer Menge von Typen deuten, welche die Funktion *len* je nach Verwendung annehmen kann:

$$\text{types}(len) = \{\ list(t) \rightarrow int \mid t \in T\ \}$$

Im Hinblick auf die Flexibilität und Bequemlichkeit der Notation wäre es wünschenswert, für arithmetische Operationen, die sowohl auf ganzen als auch reellen Zahlen definiert sind, identische Symbole zu verwenden; also beispielsweise + für die Addition von Zahlen. Dieses Vorgehen nennt man üblicherweise Überladung von Operationssymbolen. Leider läßt sich dem Symbol + nun kein eindeutiger parametrisch polymorpher Typ mehr zuordnen: Die Menge der möglichen, d.h. zulässigen Typen der Additionsoperation ist nämlich

$$\text{types}(+) = \{int \rightarrow int \rightarrow int, real \rightarrow real \rightarrow real\}$$

Wie man sich leicht überzeugt, beschreibt der Ausdruck

$$\forall t.t \rightarrow t \rightarrow t$$

eine Obermenge von types(+); andererseits führt jede Ersetzung von t durch einen der beiden zulässigen Typen *int* bzw. *real* zum Ausschluß einer zulässigen Alternative.

Hier bieten sich drei mögliche Vorgehensweisen an: Einerseits kann man auf vollständige Sicherheit des Typsystems verzichten und den Typ der Additionsoperation durch

den Typausdruck $\forall t.t \to t \to t$ beschreiben. Dann ist z.B. der Ausdruck $1 + \lambda x.x$ typisierbar. Oder man faßt die Datentypen *int* und *real* zu einem einzigen Zahldatentyp *num* zusammen, wie beispielsweise in Miranda. Damit bleibt zwar das Typsystem sicher und flexibel, man verliert jedoch einiges an Aussagekraft: die Unterscheidung von ganzen Zahlen und Fließkommazahlen erscheint uns essentiell und sollte daher im Typsystem jeder „streng“ typisierten Programmiersprache reflektiert werden.

Als dritte Möglichkeit könnte der Typisierungsalgorithmus so erweitert werden, daß er eine Menge zulässiger Typen errechnet. Falls die Menge leer ist, liegt ein Typfehler vor, ist sie mehrelementig, dann spricht man von einer nicht auflösbaren Überladung. Dieser Ansatz wurde z.B. in HOPE und Standard ML verfolgt; er bildet auch die Grundlage der Typinferenzkomponente des PSG-Systems [BS86, Sne91, SB89]. Er hat jedoch den entscheidenden Nachteil, daß überladene Operatoren nicht polymorph verwendet werden können: So hat beispielsweise der Ausdruck $\lambda x.x + 3$ den Typ *int*, der Ausdruck $\lambda x.x + x$ dagegen ist nicht typisierbar, da er sowohl den Typ $int \to int$ als auch $real \to real$ besitzt. Als Konsequenz daraus, kann man z.B. auch keine polymorphe Funktion definieren, die eine Liste von Zahlen addiert.

Man beachte jedoch, daß es auch einen semantischen Unterschied zwischen Überladungen und parametrischem Polymorphismus gibt: Im Fall des überladenen Operators + wäre zu erwarten, daß ein Auswertungssystem eine Ganzzahladdition anders interpretiert als eine Fließkommaaddition; dagegen erwartet man die Existenz einer uniformen Vorschrift zur Berechnung der Länge einer Liste. Mit anderen Worten: die Semantik einer Additionsoperation hängt vom Typ der Argumente ab, das Gegenteil gilt für die Funktion *len*. Aufgrund dieser Unterscheidung werden Überladungen auch gern als „ad hoc Polymorphismus“ bezeichnet [Str67]. Natürlich ist diese Unterscheidung etwas willkürlich, denn man kann sich leicht Maschinen mit einer Additionsinstruktion vorstellen, die den Typ der Argumente testet und dann entweder eine Ganzzahl- oder eine Fließpunktaddition ausführt.

Ein markantes, und überraschendes Beispiel für die Schwächen eines rein parametrisch polymorphen Typsystems ist die Vergleichsoperation: der Operator = soll auf Werte beliebigen Typs angewendet werden können, vorrausgesetzt es handelt sich bei den verglichenen Werten nicht um Funktionen, bzw. um Datenstrukturen, die als Subkomponenten Funktionen beinhalten.

Der Grund für diese Einschränkung ist offensichtlich: da die Gleichheit von Funktionen nicht entscheidbar ist, sollte das Typsystem jeden Vergleich funktionaler Werte verbieten. Mit einfach parametrisierten Typausdrücken kann man der Gleichheitsoperation nur den Typ $\forall t.t \to t \to \mathit{bool}$ zuordnen. Der Typausdruck wird dabei wie folgt interpretiert: für irgendeinen Typ t bildet $=$ zwei Argumente dieses Typs auf einen booleschen Wert ab. Da für t jeder beliebige Typ zulässig ist, erlaubt das Typsystem somit beispielsweise den Vergleich von Funktionen vom Typ $\mathit{int} \to \mathit{int}$.

An diesem Beispiel wird deutlich, daß der Versuch einer Repräsentation der möglichen Typen überladener Operatoren durch eine Menge von Typausdrücken scheitern muß: im Fall der Gleichheitsoperation wäre eine unendliche Menge von Typausdrücken notwendig:

$$\begin{aligned} &\mathit{int} \to \mathit{int} \to \mathit{bool}, \ldots \\ &\quad \mathit{list}(\mathit{int}) \to \mathit{list}(\mathit{int}) \to \mathit{bool}, \ldots \\ &\qquad \mathit{list}(\mathit{list}(\mathit{int})) \to \mathit{list}(\mathit{list}(\mathit{int})) \to \mathit{bool}, \ldots \end{aligned}$$

Da in diesem Fall die Entwicklung eines Inferenzalgorithmus zumindest sehr schwierig erscheint, folgern wir daraus, daß es notwendig ist, endliche Repräsentanten für solche unendlichen Mengen zu entwickeln.

Im ersten Teil der vorliegenden Arbeit (Kapitel 3) entwickeln wir dazu die Theorie der *parametrischen Überladungen*. Parametrische Überladungen sind dadurch gekennzeichnet, daß die Menge der Überladungsinstanzen eines überladenen Operators (1.) durch ein *Überladungstypschema* über *einer* ausgezeichneten Typvariablen beschrieben wird und (2.) die Menge der Instanzen für diese Typvariable *induktiv* über die *Struktur* von Typausdrücken beschrieben werden kann. Auf der Basis dieser Einschränkungen werden dann Typvariablen mit Mengen von Operatornamen gekennzeichnet, um die Menge der Instanztypen einer solcherart markierten Variable α_X auf die Menge der Typen einzuschränken, die als Argumente aller in X genannten Operatoren zulässig sind. Beispielsweise erhält dann der Gleichheitsoperator den Typ $\forall \alpha_{\{=\}}.\alpha_{\{=\}} \to \alpha_{\{=\}} \to \mathit{bool}$. Wir entwickeln einen Unifikationsalgorithmus für solche Typausdrücke und zeigen, daß sich alle wichtigen Eigenschaften des parametrischen Polymorphismus auf das System der parametrischen Überladungen übertragen lassen.

In einem polymorphen Typsystem, das auch Konversionen unterstützt, muß der allgemeinste Typ von Programmausdrücken relativ zu einer Menge von Subtyprestriktionen angegeben werden. So hat beispielsweise die Funktion $twice \equiv \lambda f.\lambda x.f(f(x))$ den allgemeinsten Typ $\forall\alpha,\beta.(\alpha \to \beta) \to \alpha \to \beta \mid \beta \triangleleft \alpha$, d.h. die Funktion *twice* bildet Funktionen vom Typ $\alpha \to \beta$ ab auf Funktionen vom Typ $\alpha \to \beta$, unter der Voraussetzung, daß β ein Untertyp von α ist. Also ist beispielsweise *twice* f ein zulässiger Ausdruck, falls f vom Typ $real \to int$ ist, da $int \triangleleft real$ eine gültige Konversion ist. Damit ist klar, daß eine Erweiterung des Unifikationsalgorithmus für parametrische Überladungen nicht ausreicht, um zu einem Typinferenzalgorithmus zu gelangen, der Überladungen und Konversionen integriert behandelt.

Das Symbol $\triangleleft$ kann man als binäres Prädikat $p_\triangleleft$ auf der Menge der monomorphen Typausdrücke interpretieren, das die Menge der möglichen Instanzen von α und β auf Paare (s,t) einschränkt, für die $p_\triangleleft(s,t)$ wahr ist. Allgemeine überladene Operatoren können ähnlich behandelt werden: für jedes überladene Funktionssymbol o führt man ein n-stelliges Prädikatsymbol p_o ein, das die zulässigen Instanztypen der Überladungspositionen beschreibt. Der Funktion o wird dann das *eingeschränkte Typschema* $\forall\alpha_1,\ldots,\alpha_n.\tau|p_o(\alpha_1,\ldots,\alpha_n)$ zugeordnet. Der Typ des Gleichheitsoperators lautet in dieser Notation $\forall\alpha.\alpha \to \alpha \to bool|p_=(\alpha)$.

Überladungs- und Konversionsbedingungen können beliebig kombiniert werden, wie das folgende, einfache Beispiel der Funktion $inc = \lambda x.x{+}1$ zeigt: der allgemeinste Typ der Funktion *inc* lautet $\forall\alpha,\beta.\alpha \to \beta|int \triangleleft \beta, \alpha \triangleleft \beta, p_+(\beta)$ und wird folgendermassen interpretiert: die Funktion *inc* kann auf Werte vom Typ s angewendet werden und liefert dann einen Wert vom Typ t, vorausgesetzt *int* und s sind Untertypen von t und der Additionsoperator ist für Werte vom Typ t definiert.

Die Tatsache, daß Typaussagen nun relativ zu einer Menge von Bedingungen gemacht werden, führt sofort zur Frage der *Erfüllbarkeit* von Restriktionsmengen: unter einer nicht erfüllbaren Menge von Bedingungen kann man natürlich für einen gegebenen Ausdruck jeden beliebigen Typ herleiten. Somit kann man nur von einer gültigen Typisierung sprechen, wenn es eine Belegung der involvierten Typvariablen gibt, die alle Bedingungen erfüllt.

In Kapitel 4 entwickeln wir daher eine Theorie der *eingeschränkten Typen*[2]. Dazu

[2] engl. „constrained types“.

definieren wir ein generisches Typdeduktionssystem, das einen einfachen Algorithmus zur Berechnung allgemeinster Typen ermöglicht. Der Algorithmus berechnet für einen gegebenen Ausdruck M einen Typausdruck τ und eine Menge von Bedingungen C, sodaß M typisierbar ist, falls eine Substitution S existiert, die alle Bedingungen in C erfüllt. Wir führen weiter das Konzept der *vereinfachenden Substitutionen* ein und zeigen, daß man eine kanonische Repräsentation des allgemeinsten Typs auf Basis *genauester Vereinfachungen* erhält. Auf der Basis dieses Konzepts modifizieren wir den Inferenzalgorithmus für eingeschränkte Typen und erhalten den Milner'schen Algorithmus $\mathcal{W}$ für parametrischen Polymorphismus als Spezialfall.

In Kapitel 5 untersuchen wir die Frage, für welche Klassen von Prädikaten die Lösung von Restriktionsproblemen entscheidbar ist und zeigen, daß für das System struktureller Konversionen und parametrischer Überladungen, kombiniert mit nicht rekursiven Überladungen, die Restriktionsauflösung sowie eine genaueste Vereinfachung berechenbar ist.

Kapitel 6 untersucht, unter welchen Bedingung die Berechnung von Lösungen und genauesten Vereinfachungen für atomare Restriktionsmengen für parametrische Überladungen und strukturelle Konversionen effizient durchgeführt werden kann und präsentiert die entsprechenden Algorithmen. Schließlich untersuchen wir in Kapitel 7 die Erweiterung unserer Resultate auf reguläre Typen und zeigen, daß unsere Algorithmen nur für Überladungen, aber nicht für strukturelle Konversionen vollständig sind. Im abschließenden Kapitel 8 werden die wichtigsten Resultate resümiert und einige noch offene Fragestellungen erörtert.

Kapitel 2

Parametrischer Polymorphismus

In diesem Kapitel rekapitulieren wir die wichtigsten Eigenschaften des parametrischen Polymorphismus. Dabei werden überwiegend bekannte Resultate dargestellt und bewiesen, einerseits um dem nicht mit der Materie vertrauten Leser einen umfassenden Überblick zu geben, andererseits, weil sich die Resultate und Beweise im zweiten Kapitel direkt auf die parametrischen Überladungen übertragen lassen. Darüber hinaus werden wir eine Modifikation des Damas-Milner-Typsystems verwenden, die zwar schon in [CDDK86] benutzt wurde, für die die Beweise der entsprechenden Resultate nach Kenntnis des Autors aber noch nicht veröffentlicht wurden. Typinferenzspezialisten können diesen Abschnitt getrost überspringen und direkt zum dritten Kapitel voranschreiten, da bei Bedarf auf die entsprechenden Resultate bzw. Beweise zurückverwiesen wird.

Das Kapitel ist wie folgt aufgebaut: zunächst (Abschnitt 2.1) führen wir die wichtigsten Grundbegriffe ein: Ausdrücke, Typausdrücke, Substitutionen und Typdeduktionssysteme, Unifikation und Typinferenz (Unterabschnitte 2.1.1 bis 2.1.3). Wir besprechen kurz den polymorphen Lambda-Kalkül (Unterabschnitt 2.1.4), für den bedauerlicherweise bis heute unklar ist[1], ob die Frage der Typisierbarkeit entscheidbar ist. Im zweiten Teil (2.2) stellen wir die von Milner getroffenen Einschränkungen vor, die zu einem entscheidbaren Typdeduktionssystem führen (2.2.1) und präsentieren einige syntaktische Eigenschaften des Typdeduktionssystems (2.2.2) in Bezug auf die Reduktions- bzw. Konversionsregeln des typfreien Lambda-Kalküls. Anschließend stellen wir den Milner'schen Typinferenzalgorithmus $\mathcal{W}$ vor, beweisen dessen syntaktische Korrektheit und Vollständigkeit (Abschnitt 2.2.3) und erwähnen die bekannten Komplexitätsresultate (Abschnitt 2.2.4). Abschließend präsentieren wir eine Semantik für die betrachtete Beispielsprache und Typausdrücke und beweisen, daß wohltypisierte Programme typfehlerfrei sind (Abschnitt 2.2.5).

[1] Zitat Milner: "an embarassing open problem".

2.1 Typsystemgrundlagen

Gegenstand unserer Untersuchung bildet eine Ausdruckssprache, die als Teilmenge jeder funktionalen Sprache vorhanden ist. Sie ist eine syntaktische Obermenge des elementaren untypisierten Lambda-Kalküls, die neben der Abstraktion und Applikation noch die Let-Abstraktion umfaßt. Daher nennen wir diese Sprache Λ^{let}, oder auch Mini-ML, wg. der überragenden Bedeutung von ML für die Entwicklung des parametrischen Polymorphismus. Obwohl Λ^{let} natürlich keine realistische Programmiersprache ist, ist sie doch mächtig genug, sowohl die syntaktischen als auch die semantischen Aspekte parametrisch polymorpher Programmiersprachen zu beschreiben.

2.1.1 Ausdrücke, Typen, Substitutionen

Definition 2.1 (Ausdrücke). Sei $\mathbf{V} = \{x, y, \ldots\}$ eine abzählbare Menge von syntaktischen Variablen. Die Sprache der Ausdrücke $M \in \Lambda^{\mathsf{let}}$ wird durch die folgende abstrakte Syntax festgelegt:

$$M ::= x \mid \lambda x.M_1 \mid M_1\, M_2 \mid \textbf{let}\ x = M_1\ \textbf{in}\ M_2$$

Für $M \in \Lambda^{\mathsf{let}}$ ist die Menge der in M frei vorkommenden Variablen durch die Funktion $\mathcal{FV} : \Lambda^{\mathsf{let}} \to \mathbf{2}^{\mathbf{V}}$ definiert:

$$\begin{aligned}
\mathcal{FV}(x) &= \{x\} \\
\mathcal{FV}(\lambda x.M_1) &= \mathcal{FV}(M_1) - \{x\} \\
\mathcal{FV}(M_1\, M_2) &= \mathcal{FV}(M_1) \cup \mathcal{FV}(M_2) \\
\mathcal{FV}(\textbf{let}\ x = M_1\ \textbf{in}\ M_2) &= \mathcal{FV}(M_1) \cup (\mathcal{FV}(M_2) - \{x\})
\end{aligned}$$

Aufgabe der Typanalyse ist es die Typisierbarkeit von Ausdrücken zu überprüfen und den Programmierer auf eventuelle Fehler aufmerksam zu machen. Grundlage der Typisierung ist daher zunächst ein gemeinsames Verständnis des Typbegriffs.

Definition 2.2 (Typausdrücke). Sei F eine Menge von Typkonstruktoren mit Stelligkeit $\rho : F \to \mathbb{N}$. Für eine abzählbare Menge von Variablen $\mathcal{V} = \{\alpha, \beta, \ldots\}$ ist die Menge der Typausdrücke über $\mathcal{V}$ ($\tau \in T_F(\mathcal{V})$) gegeben durch die abstrakte Syntax:

$$\tau ::= \alpha \mid f(\tau_1, \ldots, \tau_{\rho(f)})$$

Anders ausgedrückt: Typausdrücke sind Elemente der von den Variablen und Funktionssymbolen frei erzeugten Termalgebra $T_F(\mathcal{V})$. Typausdrücke, die keine Variablen enthalten, also Elemente von $T_F(\emptyset)$, bezeichnen wir als monomorph, solche die Variablen enthalten, werden polymorph genannt. Polymorphe Typausdrücke können als eine Repräsentation von Typmengen aufgefaßt werden. Die Menge der Typvariablen eines Typausdrucks τ bezeichnen wir mit $\mathcal{V}(\tau)$.

Wir gehen im Folgenden davon aus, daß F die üblichen Typkonstruktoren umfaßt und auf jeden Fall den Funktionstypkonstruktor $\rightarrow$ enthält. Ein typische Auswahl ist $F = \{int, real, bool, char, list, set, \times, \rightarrow\}$.

Definition 2.3 (Substitutionen). Eine *Substitution* $S \in \mathcal{V} \rightarrow T_F(\mathcal{V})$ ist eine Abbildung von Variablen auf Typen, die sich nur an endlich vielen Stellen von der Identität unterscheidet. Substitutionen werden als Menge von Paaren $\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ geschrieben oder bei einer bekannten Indexmenge $\{\alpha_i \mapsto \tau_i\}$. Die Menge der Variablen α mit $S(\alpha) \neq \alpha$ heißt $dom(S)$, die Menge $\bigcup_{\alpha \in dom(S)} \mathcal{V}(S(\alpha))$ heißt $cod(S)$. Eine Substitution S *involviert* die Variable α, falls $\alpha \in (dom(S) \cup cod(S))$. Die Menge der von S involvierten Variablen bezeichnen wir mit $inv(S)$. Sind R, S Substitutionen, dann ist ihre Komposition $R \circ S$ gegeben durch $(R \circ S)x = R(Sx)$. Eine Substitution S heißt *idempotent*, falls $S \circ S = S$.[2]

Substitutionen werden in der üblichen Weise auf Typausdrücke homomorph fortgesetzt:

$$S^*(\alpha) = S(\alpha)$$
$$S^*(f(\tau_1, \ldots, \tau_n) = f(S^*(\tau_1), \ldots, S^*(\tau_n))$$

Im Folgenden werden wir zwischen einer Substitution S und ihrer homomorphen Fortsetzung S^* notationell keinen Unterschied machen.

Definition 2.4. Eine Menge A mit einer zweistelligen Relation $R \subseteq A \times A$ heißt *Quasiordnung*, falls R reflexiv und transitiv ist, d.h.: $(x, x) \in R$ und $(x, y) \in R \wedge (y, z) \in R \Rightarrow (x, z) \in R$ für alle $x, y, z \in A$.

Eine partielle Ordnung $R \subseteq A \times A$ ist eine antisymmetrische Quasiordnung: $(x, y) \in R \wedge (y, x) \in R \Rightarrow x = y$.

[2] Dies ist äquivalent zu $dom(S) \cap cod(S) = \emptyset$.

Definition 2.5 (Instanzen). Sei τ ein Typausdruck. Ein Typausdruck τ' ist eine *Instanz* von τ (notiert als $\tau' \leq \tau$), falls eine Substitution S existiert mit $\tau' = S(\tau)$. In diesem Fall nennen wir τ allgemeiner als τ'. Analog definiert man für Substitutionen: $R \leq S \iff \forall \alpha : R(\alpha) \leq S(\alpha)$.

Die Menge der Substitutionen bildet mit der so definierten Relation $\leq$ eine Quasiordnung.

Definition 2.6. Eine Äquivalenzrelation $\equiv \subseteq A \times A$ ist eine symmetrische Quasiordnung: $x \equiv y \Rightarrow y \equiv x$.

Für $x \in A$ wird die Äquivalenzklasse von x bzgl. $\equiv$ ist definiert durch

$$[x]_{\equiv} \stackrel{\text{def}}{=} \{\, y \in A \mid x \equiv y \,\}$$

Die Faktorisierung einer Menge A bzgl. einer Äquivalenzrelation $\equiv$ ist gegeben durch

$$A/_{\equiv} \stackrel{\text{def}}{=} \{\, [x]_{\equiv} \mid x \in A \,\}$$

Beim Übergang von einer quasi geordneten Menge $(A, \leq)$ zu ihrer Faktorisierung $(A/_{\cong}, \leq)$, wobei $x \cong y \iff x \leq y \wedge y \leq x$ und $x \leq y$ in $A/_{\cong}$ mit $\exists a \in [x]_{\cong},\ b \in [y]_{\cong} : a \leq b$ interpretiert wird, gewinnt man im allgemeinen eine partielle Ordnung. Für $T_F(\mathcal{V})$ gilt jedoch

Lemma 2.7. *Die Menge der Terme bildet bzgl. der Instanzrelation modulo $\cong$ einen oberen Halbverband $(T_F(\mathcal{V})/_{\cong}, \leq)$.*

2.1.2 Typdeduktionssysteme

Zur Beschreibung der Menge der typisierbaren Ausdrücke bieten sich nun zwei Möglichkeiten an: Einerseits könnte man einen Algorithmus A angeben, der einen gegebenen Ausdruck auf Typisierbarkeit überprüft und dabei einen, mehrere oder alle möglichen Typen eines Ausdrucks berechnet. Die Menge der typisierbaren Ausdrücke wäre dann durch $\{M \mid M \in \Lambda^{\mathsf{let}} : t \in A(M)\}$ gegeben. Dieses Vorgehen hat jedoch den entscheidenden Nachteil, daß Argumente über die Typisierbarkeit von Ausdrücken nur unter Bezugnahme auf einen konkreten Algorithmus geführt werden können und

somit stark von der gewählten Programmiernotation und den verwendeten Datenstrukturen beeinflußt werden. Die zweite Möglichkeit liegt in der Verwendung eines Deduktionsystems zur Ableitung von Aussagen der Form: unter einer Menge von Voraussetzungen A hat M den Typ t. Dies ist die Standard-Vorgehensweise, man findet sie in der umfangreichen Literatur zum Thema Typinferenz zum ersten Mal in [DM82].

Definition 2.8 (Typdeduktionssysteme). Ein *Typdeduktionssystem* R besteht aus einer Menge von Axiomen bzw. Deduktionsregeln der Form $\frac{F_1, \ldots, F_n}{F}$. Die F_i heißen Prämissen der Deduktionsregel und F ist die Konsequenz der Deduktionsregel. Axiome, d.h. Regeln der Form $\overline{F}$, werden üblicherweise kurz durch F notiert. Eine Aussage A heißt ableitbar in R, $(\vdash_R A)$, falls eine Sequenz von Aussagen $A_1, \ldots, A_n = A$ existiert, sodaß für alle $i = 1..n$ die Aussage A_i durch Anwendung einer Deduktionsregel in R aus einer Teilmenge von Aussagen $\{ A_{i_j} \mid i_j < i \}$ folgt.

Aussagen über den Typ eines Ausdrucks hängen im allgemeinen von den Typen der in ihm frei vorkommenden Variablen ab. Zur Formalisierung des Begriffs Typaussage benötigt man daher die folgende

Definition 2.9 (Typannahmen). Eine *Typannahme* Γ ist eine endliche Abbildung von Variablen auf Typausdrücke bzw. Typschemata[3]. Zur Notation von Typannahmen und den auf ihnen definierten Operationen verwenden wir eine VDM-ähnliche Schreibweise[4]:

$[x_1{:}\tau_1, \ldots, x_n{:}\tau_n]$	eine Typannahme, die der Variablen x_i den Typausdruck τ_i zuordnet.
$\Gamma(x)$	der Typausdruck τ, der x in Γ zugeordnet ist.
Γ_x	die Typannahme Γ ohne die Zuordnung für x.
$\Gamma_1 \cup \Gamma_2$	Vereinigung der Typannahmen Γ_1, Γ_2, wobei $dom(\Gamma_1) \cap dom(\Gamma_2) = \emptyset$ gelten muß.

[3] Siehe unten.
[4] Siehe [BJ78] oder [BJ82].

$\Gamma_1 + \Gamma_2$ Addition von Abbildungen:

$$(\Gamma_1 + \Gamma_2)(x) = \begin{cases} \Gamma_2(x) & \text{falls } x \in dom(\Gamma_2), \\ \Gamma_1(x) & \text{sonst.} \end{cases}$$

mithin gilt: $\Gamma + [x : \tau] = \Gamma_x \cup [x : \tau]$.

$\Gamma_{|M}$ Restriktion von Γ auf die Menge M:

$$\Gamma_{|M}(x) = \begin{cases} \Gamma(x) & \text{falls } x \in M, \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

$\mathcal{FV}(\Gamma)$ die Menge der in Γ frei vorkommenden Typvariablen:

$$\mathcal{FV}(\Gamma) = \bigcup_{x \in dom(\Gamma)} \mathcal{FV}(\Gamma(x))$$

$S(\Gamma)$ Anwendung von Substitutionen auf Typannahmen:
$S(\Gamma) = [\, x \mapsto Sx \mid x \in dom(\Gamma) \,]$.

Definition 2.10 (Typaussage). Eine *Typaussage* ist eine Aussage der Form $\Gamma \vdash M : \tau$, wobei Γ eine Typannahme über die freien Variablen von M ist: $\mathcal{FV}(M) \subseteq dom(\Gamma)$. Eine Typaussage $\Gamma \vdash M : \tau$ heißt *ableitbar*, falls sie durch Anwendung der Axiome und Deduktionsregeln bewiesen werden kann. In diesem Fall sprechen wir auch von einer gültigen Typaussage.

Beschränkt man sich zunächst auf let-freie Ausdrücke, dann genügt das in Abbildung 2.1 angegebene Deduktionssystem zur Beschreibung der typisierbaren Ausdrücke. Dieses Deduktionssystem besitzt die bemerkenswerte Eigenschaft, daß jeder Ableitungsbaum für einen bestimmten Ausdruck die Struktur des Ausdrucks besitzt; die Anwendung der Deduktionsregeln ist somit vollständig syntaxgesteuert.

2.1.3 Unifikation, Typinferenz

Aufgabe der Typinferenz ist die Konstruktion einer ableitbaren Typaussage. Grundlage der Beweiskonstruktion ist dabei in der Regel ein Unifikationsalgorithmus[5], mit

[5] oder ein Constraint-Solver, siehe Abschnitt 5.

$$[\text{VAR}] \quad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$[\text{ABS}] \quad \frac{\Gamma + [x : \tau_1] \vdash M : \tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \to \tau_2}$$

$$[\text{APP}] \quad \frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \;,\; \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1 M_2 : \tau_2}$$

Abbildung 2.1: Deduktionssystem für let-freie Ausdrücke

dessen Hilfe die durch das Deduktionssystem implizierten Gleichungen gelöst werden. Zur Definition des Typinferenzalgorithmus für die obige Beispielsprache benötigen wir das bekannte Unifikationstheorem von Robinson.

Definition 2.11 (Unifikatoren). Seien x, y Elemente einer freien Termalgebra. Ein *Unifikator* von x und y ist eine Substitution U mit $U(x) = U(y)$. Ein allgemeinster Unifikator von x, y ist ein Unifikator U sodaß für jeden anderen Unifikator U' eine Substitution S existiert mit $U' = S \circ U$.

Satz 2.12 (Robinson-Unifikation). *[Rob65] Eine Menge von Termen hat entweder keinen Unifikator, oder einen allgemeinsten. Es existiert ein Algorithmus ($\mathcal{U}$), der den allgemeinsten Unifikator einer Menge von Termen berechnet, oder mit einer Fehlermeldung anhält, falls kein Unifikator existiert. Der allgemeinste Unifikator ist idempotent und involviert nur Variablen, die in der zu unifizierenden Termmenge vorkommen.*

Die Erweiterung des Begriffs „allgemeinster Unifikator“ auf Mengen von Termen, Termpaare und Mengen von Termpaaren ist offensichtlich. Für endliche Abbildungen m_1, m_2, z.B. Typannahmen, ist U ein Unifikator, falls $U(m_1(x)) = U(m_2(x))$ für alle $x \in dom(m_1) \cap dom(m_2)$.

Der hier in der Originalversion angegebene Unifikationsalgorithmus hat die unangenehme Eigenschaft zur Berechnung des allgemeinsten Unifikators u.U. exponentielle Rechenzeit zu benötigen und als Ergebnis Terme mit exponentiellem Platzbedarf zu

Algorithmus 2.1. ($\mathcal{U}$) Robinson Unifikation in freien Termalgebren

$$
\begin{array}{ll}
\mathcal{U}(\tau,\tau) = \{\} & \\
\mathcal{U}(\tau,\alpha) = \mathcal{U}(\alpha,\tau) & \text{falls } \tau \notin \mathcal{V} \\
\mathcal{U}(\alpha,\tau) = \{\alpha \mapsto \tau\} & \text{falls } \alpha \notin \mathcal{V}(\tau) \\
\mathcal{U}(c(\tau_1,\ldots,\tau_n), c(\tau'_1,\ldots,\tau'_n)) = S_n & \text{falls } \exists S_0,\ldots,S_n. \\
 & \quad S_0 = \{\} \wedge \forall i = 1..n : \\
 & \quad\quad S_i = \mathcal{U}(S_{i-1}(\tau_i), S_{i-1}(\tau'_i)) \circ S_{i-1}
\end{array}
$$

In allen anderen Fällen schlägt der Algorithmus fehl

erzeugen. Als Beispiel betrachte man das Unifikationsproblem[6]

$$
\begin{array}{l}
\mathcal{U}(\ g(\ \ a_1,\ \ a_2,\ \ \ldots,\ \ a_n,\ \ b_1,\ \ \ldots,\ \ b_n,\ \ a_n\ ), \\
\quad g(\ f(a_0,a_0),\ f(a_1,a_1),\ \ldots,\ f(a_{n-1},a_{n-1}),\ f(b_0,b_0),\ \ldots,\ f(b_{n-1},b_{n-1}),\ b_n\ ))
\end{array}
$$

Da die Ergebnisterme exponentiell viele Teilterme enthalten, muß man von der Termdarstellung zu einer Darstellung der Terme durch gerichtete (azyklische) Graphen übergehen. In der einfachsten Implementierung erhält man quadratischen Aufwand [CB83]. Verwendet man jedoch zusätzlich eine Variante des UNION-FIND-Algorithmus[7] [MM82], dann erhält man quasi-linearen Aufwand. Ein vollständig linearer Algorithmus wurde von Paterson und Wegman in [PW78] angegeben. Wir werden später in Abschnitt 3.4.3 noch auf die konkrete Implementierung im SAMPλE-System zurückkommen.

Das Typdeduktionssystem für let-freie Ausdrücke ermöglicht eine einfache „bottom-up“ Implementierung der Typinferenz: die Regeln [VAR] und [ABS] erzwingen, daß jeder Verwendung eines Bezeichners der gleiche Typausdruck zugeordnet wird.

Satz 2.13. *Jeder typisierbare Ausdruck besitzt eine allgemeinste Typisierung* $\Gamma \vdash M : \tau$*, sodaß für jede andere Typisierung* $\Gamma' \vdash M : \tau'$ *eine Substitution existiert, mit* $\Gamma' = S$ *und* $\tau' = S\tau$*.*

Wir geben den Algorithmus ohne Beweis an, für den der Leser an [Let86] verwiesen wird.

[6] Entnommen aus [Nip].
[7] Siehe [AHU74, Kapitel 4.7].

Algorithmus 2.2. ($\mathcal{Z}$) Typinferenz für einfachen Polymorphismus

$\mathcal{Z}[\![x]\!] = (\beta, [x : \beta])$
wobei β eine „neue“, bisher noch nicht verwendete Typvariable ist.

$\mathcal{Z}[\![M_1\, M_2]\!] = (U_2\beta, U_2U_1\Gamma_1 + U_2U_1\Gamma_2)$
Falls Typannahmen Γ_1, Γ_2 und Typen τ_1, τ_2 existieren, sodaß
$(\tau_1, \Gamma_1) = \mathcal{Z}[\![M_1]\!]$
$(\tau_2, \Gamma_2) = \mathcal{Z}[\![M_2]\!]$
$U_1 = \mathcal{U}(\Gamma_1, \Gamma_2)$
$U_2 = \mathcal{U}(U_1\tau_1, U_1\tau_2 \to \beta)$
und β eine „neue“ Typvariable ist.

$\mathcal{Z}[\![\lambda x.M]\!] = (\Gamma(x) \to \tau, \Gamma_x)$
Falls $x \in \mathcal{FV}(M)$,
$(\tau, \Gamma) = \mathcal{Z}[\![M]\!]$
und β eine „neue“ Typvariable ist.

$\mathcal{Z}[\![\lambda x.M]\!] = (\beta \to \tau, \Gamma)$
Falls $x \notin \mathcal{FV}(M)$,
$(\tau, \Gamma) = \mathcal{Z}[\![M]\!]$
und β eine „neue“ Typvariable ist.

In allen anderen Fällen schlägt der Algorithmus fehl.

2.1.4 Der polymorphe Lambda-Kalkül

Das Typdeduktionssystem für let-freie Ausdrücke gestattet zwar die Berechnung allgemeinster Typen, jedoch besteht für den Programmierer keine Möglichkeit solcherart typisierte Ausdrücke auch polymorph zu verwenden. So hat zwar der Ausdruck $\lambda x.x$ den allgemeinsten Typ $\alpha \to \alpha$, doch der Ausdruck $(\lambda y.yy)(\lambda x.x)$ ist mit dem obigen System nicht typisierbar[8]: Der Parameter y darf im Körper der Abstraktion $\lambda y.yy$ nicht polymorph verwendet werden. Mit anderen Worten: der Polymorphismus ist auf die Meta-Ebene beschränkt.

[8] Jedenfalls nicht, wenn nur endliche Typen erlaubt sind (siehe Kapitel 7).

$$[\text{VAR}] \quad \frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma}$$

$$[\text{ABS}] \quad \frac{\Gamma + [x : \sigma_1] \vdash M : \sigma_2}{\Gamma \vdash \lambda x.M : \sigma_1 \to \sigma_2}$$

$$[\text{APP}] \quad \frac{\Gamma \vdash M_1 : \sigma_1 \to \sigma_2 \, , \, \Gamma \vdash M_2 : \sigma_1}{\Gamma \vdash M_1 M_2 : \sigma_2}$$

$$[\text{GEN}] \quad \frac{\Gamma \vdash M : \sigma \, , \, \alpha \notin FV(\Gamma)}{\Gamma \vdash M : \forall \alpha.\sigma}$$

$$[\text{SPEC}] \quad \frac{\Gamma \vdash M : \forall \alpha.\sigma}{\Gamma \vdash M : \sigma[\sigma'/\alpha]}$$

Abbildung 2.2: Typdeduktion im polymorphen Lambda-Kalkül

Um diese Einschränkung aufzuheben, und den Polymorphismus auf Objekt-Ebene zu bringen, wird die Menge der Typen um die Möglichkeit erweitert, Variablen als unabhängig vom Kontext substituierbar zu kennzeichnen. Die solcherart erweiterten Typausdrücke werden von der folgenden abstrakten Syntax generiert.

$$\sigma ::= \alpha \mid f(\sigma_1, \ldots, \sigma_{\rho(f)}) \mid \forall \alpha.\sigma$$

Damit steht $\forall \alpha.\alpha \to \alpha$ für einen Typ, in dem α unabhängig vom Kontext durch beliebige Typen ersetzt werden kann, und $\alpha \to \alpha$ bezeichnet einen Funktionstyp, der zwar unbekannt ist, i.d.R. aber durch den Kontext festgelegt wird. Man beachte, daß mit dieser Typsyntax auch Typausdrücke möglich sind, bei denen der Abstraktionsoperator $\forall$ nicht auf der obersten Ebene verwendet wird. So beschreibt beispielsweise der Typausdruck $\forall \alpha.(\forall \beta.\beta \to \beta) \to \alpha \to \alpha$ eine Funktion, die eine polymorphe Funktion vom Typ $\forall \beta.\beta \to \beta$ als erstes Argument akzeptiert, und auf eine Funktion vom Typ $\alpha \to \alpha$ abbildet.

Unter Verwendung der erweiterten Typausdrücke kann man ein Deduktionssystem definieren, das die polymorphe Verwendung von beliebigen Teilausdrücken in der

o.a. Beispielsprache erlaubt (siehe Abbildung 2.2). Es enthält 2 neue Regeln, die die Verallgemeinerung bzw. die Spezialisierung von Typausdrücken festlegen.[9] Dieses Deduktionssystem ist im Gegensatz zu dem in Abbildung 2.1 angegebenen Deduktionssystem nicht syntaxorientiert: die Deduktionsregeln [GEN] bzw. [SPEC] sind nicht an bestimmte syntaktische Konstrukte gebunden, sondern können an jeder beliebigen Stelle in der Sequenz der Ableitungschritte eingesetzt werden. Dies hat den Vorteil, die Leistungsfähigkeit des Beweiskalküls zu erhöhen, erschwert jedoch die Konstruktion effektiver Typinferenzalgorithmen.

Dieses Typsystem ist unter mehreren Namen bekannt: System F, Lambda-Kalkül zweiter Ordnung oder auch polymorpher Lambda-Kalkül. Für eine eingehende Diskussion verweisen wir auf [Bar92].

Der polymorphe Lambda-Kalkül hat vom Standpunkt der Typinferenz aus betrachtet einen entscheidendenden Nachteil: es ist bis heute unklar, ob es für dieses Typsystem einen Typisierungsalgorithmus gibt, der feststellt, ob ein Ausdruck typisierbar ist und gegebenfalls eine Typisierung berechnet.[10] Allerdings wird weithin vermutet, daß das Typinferenzproblem nicht entscheidbar ist.

2.2 ML-Polymorphismus

Von Milner wurde in [Mil78] und später in [DM82] zwei Einschränkungen des polymorphen Lambda-Kalküls vorgeschlagen[11], die das Konzept des Polymorphismus für den Entwurf und die Implementierung von Typsystemen für reale Programmiersprachen nutzbar machten und auch für die Programmiersprache ML des LCF-Systems implementiert. Zum Einen wurde auf Polymorphismus höherer Ordnung verzichtet, d.h. der Abstraktionsoperator $\forall$ wurde auf die äußerste Ebene von Typausdrücken beschränkt. Zum Anderen wurde die Verwendung der GEN-Regel in Essenz an ein

[9] $\sigma[\sigma'/\alpha]$ bezeichnet dabei die Ersetzung der Variablen α in σ durch σ'. Dabei müssen u.U. in σ durch Abstraktionsoperatoren gebundene Variablen umbenannt werden, um die irrtümliche Bindung freier Typvariablen in σ' zu vermeiden. Dies entspricht der α-Konversion im typfreien Lambda-Kalkül

[10] Die Schwierigkeit liegt in der algorithmischen Behandlung der beiden neuen Regeln: es ist völlig unklar, wann eine der beiden Regeln in einer Deduktion angewendet werden muß, um den Gesamtausdruck zu typisieren.

[11] Auch wenn dies in den beiden Veröffentlichungen nicht explizit so konstatiert wurde.

$$[\text{LET}] \quad \frac{\Gamma \vdash M_1 : \tau_1 \ , \ \Gamma \vdash M_2[M_1/x] : \tau_2}{\Gamma \vdash \textbf{let } x = M_1 \textbf{ in } M_2 : \tau_2}$$

Abbildung 2.3: Let-Typisierung durch Let-Reduktion

spezielles syntaktisches Konstrukt gebunden: das let-Konstrukt.

Das Konstrukt **let** $x = M_1$ **in** M_2 kann als alternative Schreibweise für den Ausdruck $(\lambda x.M_2)M_1$ betrachtet werden. Hinsichtlich der Typisierung werden beide Konstrukte aber völlig unterschiedlich behandelt. Während im Fall der Lambda-Abstraktion der Typ von x für die Typisierung des Unterausdrucks M_2 fixiert wird, darf x für die Typisierung von M_2 polymorph verwendet werden.

So darf etwa der in dem let-Konstrukt

$$\textbf{let } i = (\lambda x.x) \textbf{ in } i \ i$$

definierte Bezeichner i im Rumpf der let-Abstraktion einmal mit dem Typ $(\alpha \to \alpha) \to (\alpha \to \alpha)$ und einmal mit dem Typ $\alpha \to \alpha$ verwendet werden, wohingegen der semantisch äquivalente Ausdruck $(\lambda i.i\,i)(\lambda x.x)$ nicht typisierbar ist.

Interessanterweise kann die Charakteristik des let-Konstrukts auf einfache Weise beschrieben werden: Ein Ausdruck ist typisierbar genau dann, wenn er nach Elimination aller let-Konstrukte im o.a. Deduktionssystem für let-freie Ausdrücke typisierbar ist (siehe [Let83, KM89, Oho89], bzw. Abbildung 2.3).

Dabei ist die Let-Reduktion in der üblichen Weise als transitiver Abschluß der Reduktionsregel

$$\textbf{let } x = M_1 \textbf{ in } M_2 \ \longrightarrow_{let} \ M_2[M_1/x]$$

unter Abstraktion und Applikation definiert (siehe auch Abschnitt 2.2.2). Die Reduktionsrelation $\longrightarrow_{let}$ ist Church-Rosser und noethersch, d.h., jede Sequenz von let-Reduktionen führt zu einer eindeutig bestimmten, let-freien Normalform [KM89]. Vom praktischen Standpunkt aus betrachtet hat diese Methode jedoch einige Nachteile: die Größe des expandierten Terms kann exponentiell wachsen, die Definition

bietet wenig Anhaltspunkte für einen konkreten Typinferenzalgorithmus und Typableitungen sind selbst für relativ einfache Beispielprogramme nur schwer von Hand durchzuführen.

2.2.1 Typsyntax und Typdeduktion

Die Einschränkung der polymorphen Typausdrücke auf Pränex-Quantifizierung erreicht man durch eine geringfügige Modifikation der abstrakten Syntax. Diese Typausdrücke werden in der Literatur üblicherweise als Typschemata bezeichnet.

Definition 2.14 (Typschemata). Die Menge der *Typschemata* ist gegeben durch die abstrakte Syntax

$$\sigma ::= \tau \mid \forall\alpha.\sigma$$

Die Variablen $\alpha_1, \ldots, \alpha_n$ des Typschemas $\forall\alpha_1.\cdots\forall\alpha_n.\tau$ heißen generisch bzw. gebunden, alle anderen Typvariablen des Typschemas nennen wir spezifisch. Da der Variablenabstraktor $\forall$ nur auf der äußersten Ebene eines Typschemas auftreten kann, wird $\forall\alpha_1.\cdots\forall\alpha_n.\tau$ auch abkürzend durch $\forall\alpha_1, \ldots, \alpha_n.\tau$ bzw. $\forall\overline{a_n}.\tau$ notiert. Die freien Variablen eines Typschemas bezeichnen wir in Analogie zu der Menge der freien Variablen in Ausdrücken mit $\mathcal{FV}(\sigma)$, die generischen mit $\mathcal{BV}(\sigma)$. Typschemata, die sich nur im Namen ihrer gebundenen Variablen unterscheiden, werden identifiziert. Falls α in σ nicht frei vorkommt, ist $\forall\alpha.\sigma$ äquivalent zu σ.

Ein Typausdruck τ steht für einen durch den Kontext festgelegten Typ. Der Abstraktionsoperator $\forall$ kennzeichnet die Variablen eines Typschemas, die unabhängig vom Kontext durch beliebige andere Typen bzw. Typausdrücke ersetzt werden dürfen.

Definition 2.15 (Generische Instanz von Typausdrücken und Typannahmen). Sei $S = \{\alpha_i \mapsto \tau_i\}$ eine Substitution. Für ein Typschema $\sigma = \forall\overline{\alpha_n}.\tau$ bezeichnet $S\sigma$ das Typschema σ', das man durch Instanzierung der freien Vorkommen der α_i in τ durch τ_i erhält. Dabei müssen u.U. die in σ gebundenen Variablen umbenannt werden, um Namenskollisionen zu vermeiden. Ein Typschema $\sigma' = \forall\overline{\beta_m}.\tau'$ heißt *generische Instanz* eines Typschemas $\sigma = \forall\overline{\alpha_n}.\tau$, geschrieben $\sigma' \prec \sigma$, falls $\tau' = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}\tau$ und keines der β_i kommt frei in τ vor. Für Typannahmen Γ, Γ' definieren wir $\Gamma \prec \Gamma' \iff \forall x \in dom(\Gamma).\, \Gamma(x) \prec \Gamma'(x)$.

Die Verbandseigenschaften der Relation $\leq$ übertragen sich auf die Relation $\prec$.

Satz 2.16. *Die Menge der Typschemata über $T_F(\mathcal{V})$ bildet mit der Relation $\prec$ einen Halbverband.*

Beweis. Daß $\prec$ reflexiv, transitiv und antisymmetrisch ist, folgt unmittelbar aus der Definition. Um zu sehen, daß jede Menge von Typschemata eine kleinste obere Schranke besitzt, betrachten wir eine Bijektion $(\sharp)$ von $\mathcal{V} \to T_F(\emptyset)$, die jeder Variablen α eineindeutig einen Typkonstruktor $\alpha^\sharp \in F_0$ zuordnet. Für Typschemata $\sigma = \forall\overline{\alpha_n}.\tau$ sei $\sigma^\sharp$ der Typausdruck, den man durch Elimination des Variablenabstraktors $\forall\overline{\alpha_n}$ und Substitution der freien Typvariablen in τ durch ihr Bild in $T_F(\emptyset)$ erhält. Dann gilt $\sigma_1 \prec \sigma_2 \iff \sigma_1^\sharp \leq \sigma_2^\sharp$ und damit $\sigma_1 \sqcap \sigma_2 = (\sigma_1^\sharp \sqcap \sigma_2^\sharp)^\flat$, wobei $\flat$ die Umkehrabbildung $\sharp^{(-1)}$ ist.[12] □

Darüber hinaus besitzt die Relation $\prec$ weitere Eigenschaften.

Proposition 2.17. *Seien σ_1, σ_2 Typschemata und S eine Substitution. Dann gilt:*

$$\sigma_1 \prec \sigma_2 \Rightarrow S\sigma_1 \prec S\sigma_2 \tag{i}$$

$$\sigma_1 \prec \sigma_2 \Rightarrow \mathcal{FV}(\sigma_1) \supseteq \mathcal{FV}(\sigma_2) \tag{ii}$$

Beweis. (i) folgt aus der α-Konvertierbarkeit von Typschemata, womit $inv(S) \cap (\mathcal{BV}(\sigma_1) \cup \mathcal{BV}(\sigma_2)) = \emptyset$ erfüllbar ist und (ii) direkt aus der Definition von $\prec$. □

Das in Abbildung 2.4 angegebene Typsystem von Damas-Milner ist ähnlich wie das Deduktionssystem für den polymorphen Lambda-Kalkül nicht syntaxorientiert. Bei näherer Betrachtung stellt man jedoch fest, daß die Anwendung der problematischen Regeln nur an ganz bestimmten Stellen einer Ableitung sinnvoll ist:

- Die Regeln [APP] und [ABS] sind nur auf Typen, nicht auf Typschemata anwendbar, sodaß die [GEN] Regel nicht zur Ableitung der Prämissen dieser Regeln anwendbar ist.

[12] Einen Algorithmus zur Berechnung der kleinsten oberen Schranke findet man beispielsweise in [Let86, Seite 108].

$$\text{[VAR]} \quad \frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma}$$

$$\text{[ABS]} \quad \frac{\Gamma + [x : \tau_1] \vdash M : \tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \to \tau_2}$$

$$\text{[APP]} \quad \frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \;,\; \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1 M_2 : \tau_2}$$

$$\text{[LET]} \quad \frac{\Gamma \vdash M_1 : \sigma \;,\; \Gamma + [x : \sigma] \vdash M_2 : \tau}{\Gamma \vdash \textbf{let } x = M_1 \textbf{ in } M_2 : \tau}$$

$$\text{[GEN]} \quad \frac{\Gamma \vdash M : \sigma \;,\; \alpha \notin FV(\Gamma)}{\Gamma \vdash M : \forall \alpha.\sigma}$$

$$\text{[SPEC]} \quad \frac{\Gamma \vdash M : \sigma \;,\; \sigma' \prec \sigma}{\Gamma \vdash M : \sigma'}$$

Abbildung 2.4: Das Deduktionssystem von Damas-Milner

- Nach der Anwendung der [VAR]-Regel muß sofort die [SPEC]-Regel angewendet werden, wenn danach die [ABS] oder [APP]-Regel angewendet werden soll.

- Nur die Let-Regel gestattet es, ein inferiertes Typschema zu verwenden. Daher kann [GEN] nur sinnvoll in der Prämisse zur LET-Regel und am Ende einer Ableitung angewendet werden.

Insgesamt wird dadurch die Verwendung des in Abbildung 2.5 angegebenen Deduktionssystems nahegelegt, für das wir den Begriff der Generalisierung eines Typausdrucks im Kontext einer Typannahme benötigen.

Definition 2.18 (Generalisierung von Typschemata im Kontext einer Typannahme). Sei Γ eine Typannahme und σ ein Typschema. Die Generalisierung von σ im Kontext von Γ ist das Typschema $\text{gen}(\Gamma, \sigma)$, das man durch Binden aller

$$[\text{VAR}] \quad \frac{\tau \prec \Gamma(x)}{\Gamma \vdash x : \tau}$$

$$[\text{ABS}] \quad \frac{\Gamma + [x : \tau_1] \vdash M : \tau_1}{\Gamma \vdash \lambda x.M : \tau_1 \to \tau_2}$$

$$[\text{APP}] \quad \frac{\Gamma \vdash M_1 : \tau_2 \to \tau_1 \ , \ \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash M_1 M_2 : \tau_1}$$

$$[\text{LET}] \quad \frac{\Gamma \vdash M_1 : \tau_1 \ , \ \Gamma + [x : \text{gen}(\Gamma, \tau_1)] \vdash M_2 : \tau_2}{\Gamma \vdash \textbf{let } x = M_1 \textbf{ in } M_2 : \tau_2}$$

Abbildung 2.5: Ein syntaxorientiertes Deduktionssystem für Let-Polymorphismus

Typvariablen erhält, die in σ aber nicht in Γ frei vorkommen:

$$\begin{aligned} \text{gen}(\Gamma, \sigma) &\stackrel{\text{def}}{=} \forall \alpha_1, \ldots, \alpha_n.\sigma \\ &\text{wobei } \ \{\alpha_1, \ldots, \alpha_n\} = \mathcal{FV}(\sigma) - \mathcal{FV}(\Gamma) \end{aligned}$$

Man beachte, daß wir die Generalisierung für Typschemata allgemein definiert haben; in unserem Typdeduktionssystem werden aber immer nur Typausdrücke generalisiert.

Proposition 2.19 (Eigenschaften von $\text{gen}(\Gamma, \sigma)$**).**

$$\sigma' \prec \sigma \Rightarrow \text{gen}(\Gamma, \sigma') \prec \text{gen}(\Gamma, \sigma) \tag{i}$$

$$\Gamma' \prec \Gamma \Rightarrow \text{gen}(\Gamma', \sigma) \prec \text{gen}(\Gamma, \sigma) \tag{ii}$$

$$\text{gen}(S\Gamma, S\tau) \prec S(\text{gen}(\Gamma, \tau)) \tag{iii}$$

$$inv(S) \cap (\mathcal{V}(\tau) - \mathcal{FV}(\Gamma)) = \emptyset \Rightarrow \text{gen}(S\Gamma, S\tau) = S(\text{gen}(\Gamma, \tau)) \tag{iv}$$

Beweis. (i), (ii) folgen aus Proposition 2.17(ii).

Für (iii) sei $\sigma' = \forall \overline{\beta_m}.S\tau = \text{gen}(S\Gamma, S\tau)$ und $\sigma = \forall \overline{\alpha_n}.\tau = \text{gen}(\Gamma, \tau)$. Dann existiert eine Variablenumbenennung R mit $dom(R) = \{\overline{\alpha_n}\}$ und $rng(R) \cap inv(S) = \emptyset$, sodaß $\sigma = \forall R(\overline{\alpha_n}).R\tau$ und damit $S\sigma = \forall R(\overline{\alpha_n}).S(R\tau)$. Zu zeigen ist, daß eine Substitution

Q existiert, sodaß $S\tau = Q(S(R\tau))$ und $dom(Q) = rng(R)$. Wähle $Q(R\alpha) = S\alpha$. Es gilt $\forall\alpha \in \mathcal{V}(\tau) : S\alpha = Q(S(R\alpha))$, denn: falls $\alpha \in \mathcal{FV}(\Gamma)$ gilt $\alpha \notin inv(R)$ und daher $Q(S(R\alpha)) = Q(S\alpha)$. Da $dom(Q) = rng(R)$ und $rng(R) \cap inv(S) = \emptyset$, folgt $Q(S\alpha) = S\alpha$. Für $\alpha \notin \mathcal{FV}(\Gamma)$ gilt $\alpha \in dom(R)$ und $R\alpha \notin dom(S)$. Dies impliziert $Q(S(R\alpha)) = Q(R\alpha)$, was nach Konstruktion von Q aber $S\alpha$ ist.

Für (iv) folgt wg. $\forall\overline{\beta_m}.S\tau = S(\forall\overline{\beta_m}.\tau)$ sofort die Behauptung. □

Das syntaxorientierte Deduktionssystem unterscheidet sich vom Damas-Milner System in zwei Aspekten: Instanzierung von Typschemata ($\prec$) ist auf die [VAR]-Regel eingeschränkt und Generalisierung von Typen ($\text{gen}(\Gamma, \tau)$) nur für let-eingeführte Bezeichner gestattet. Dieses System wurde erstmals in [CDDK86] verwendet. Dort wurde auch gezeigt, daß es zu dem ursprünglichen System von Damas-Milner im folgenden Sinne äquivalent ist: Jede in diesem System ableitbare Typaussage kann auch im Damas-Milner System abgleitet werden; und für jede im Damas-Milner System ableitbare Typaussage $\Gamma \vdash M : \sigma$ gibt es ein gültige Typisierung $\Gamma \vdash M : \tau$ mit der Eigenschaft $\sigma \prec \text{gen}(\Gamma, \tau)$.

Proposition 2.20 (Typisierungen sind abgeschlossen unter Substitutionen). *Falls $\Gamma \vdash M : \tau$ eine gültige Typisierung ist und S eine Substitution, dann ist auch $S\Gamma \vdash M : S\tau$ eine gültige Typisierung.*

Beweis. Durch Induktion über M. Für $M \equiv x \in V$ gilt $\Gamma \vdash M : \tau$ genau dann, wenn $\tau \prec \Gamma(x)$. Aus Proposition 2.17(i) folgt $S\tau \prec S(\Gamma(x))$ und damit ist $S\Gamma \vdash M : S\tau$ eine gültige Typisierung. Für Abstraktionen und Applikationen folgt die Aussage direkt aus der Induktionshypothese. Sei also $\Gamma \vdash \textbf{let } x = M_1 \textbf{ in } M_2 : \tau$ eine gültige Typisierung. Dann existieren Typisierungen $\Gamma \vdash M_1 : \tau_1$ und $\Gamma + [x : \text{gen}(\Gamma, \tau_1)] \vdash M_2 : \tau$. Sei R eine Variablenumbenennung mit $dom(R) = \mathcal{V}(\tau_1) - \mathcal{FV}(\Gamma)$ und $cod(R) \cap inv(S) = \emptyset$. Aufgrund der Induktionshypothese ist $S(R\Gamma) \vdash M_1 : S(R\tau_1)) = S\Gamma \vdash M_1 : S(R\tau_1)$ eine gültige Typisierung und es gilt $inv(S) \cap (\mathcal{V}(R(\tau_1)) - \mathcal{FV}(\Gamma)) = \emptyset$. Damit folgt aus Proposition 2.19(iv): $\text{gen}(S\Gamma, SR\tau_1) = S(\text{gen}(\Gamma, R\tau_1))$. Außerdem gilt $\text{gen}(\Gamma, R\tau_1) = \text{gen}(\Gamma, \tau_1)$ und da $\Gamma + [x : \text{gen}(\Gamma, \tau_1)]) \vdash M_2 : \tau$ eine gültige Typisierung ist, können wir die Induktionshypothese anwenden und erhalten gültige Typisierungen $S(\Gamma + [x : \text{gen}(\Gamma, \tau_1)]) \vdash M_2 : S\tau$ und $S\Gamma \vdash \textbf{let } x = M_1 \textbf{ in } M_2 : S\tau$. □

Proposition 2.21 (Typisierung unter Abschwächung von Typannahmen). *Sei* $\Gamma' \prec \Gamma$ *und* $\Gamma' \vdash M : \tau$ *eine gültige Typisierung. Dann ist auch* $\Gamma \vdash M : \tau$ *eine gültige Typisierung.*

Beweis. Durch Induktion über M. Falls $M \equiv x \in V$ gilt $\tau \prec \Gamma'(x)$ und damit aufgrund der Transitivität von $\prec$ auch $\tau \prec \Gamma(x)$, sodaß $\Gamma \vdash x : \tau$ eine gültige Typisierung ist. Für Abstraktionen und Applikationen folgt die Aussage direkt aus der Induktionshypothese. Sei also $\Gamma' \vdash \textbf{let } x = M_1 \textbf{ in } M_2 : \tau$ eine gültige Typisierung. Dann existieren Typisierungen $\Gamma' \vdash M_1 : \tau_1$ und $\Gamma' + [x : \mathrm{gen}(\Gamma', \tau_1)] \vdash M_2 : \tau$. Aufgrund der Induktionshypothese gilt auch $\Gamma \vdash M_1 : \tau_1$. Aus Proposition 2.19(ii) folgt $\mathrm{gen}(\Gamma', \tau_1) \prec \mathrm{gen}(\Gamma, \tau_1)$, sodaß auch $\Gamma' + [x : \mathrm{gen}(\Gamma', \tau_1)] \prec \Gamma + [x : \mathrm{gen}(\Gamma, \tau_1)]$ gilt, und damit kann die Induktionshypothese angewendet werden, um $\Gamma + [x : \mathrm{gen}(\Gamma, \tau_1)] \vdash M_2 : \tau$ zu erhalten, woraus die Behauptung folgt. □

2.2.2 Reduktion und Typisierung

Die folgenden Definitionen wurden [Bar81] entnommen und um entsprechende Klauseln für let-Ausdrücke erweitert.

Definition 2.22. Ein *Reduktionsbegriff* $\mathbf{R}$ ist eine binäre Relation auf Λ^{let}. Sind $\mathbf{R}_1$, $\mathbf{R}_2$ Reduktionsbegriffe dann bezeichnet $\mathbf{R}_1\mathbf{R}_2$ den Reduktionsbegriff $\mathbf{R}_1 \cup \mathbf{R}_2$.

Die wohl bekanntesten Reduktionsbegriffe sind die β- und η-Reduktion, die wir um die let-Reduktion ergänzen:

$$\begin{aligned}
\boldsymbol{\beta} &\stackrel{\text{def}}{=} \{\, ((\lambda x.M)N, M[N/x]) \mid M, N \in \Lambda^{\mathsf{let}} \,\} \\
\boldsymbol{\eta} &\stackrel{\text{def}}{=} \{\, (\lambda x.Mx, M) \mid M \in \Lambda^{\mathsf{let}}, x \notin M \,\} \\
\mathbf{let} &\stackrel{\text{def}}{=} \{\, (\textbf{let } x = M \textbf{ in } N, N[M/x]) \mid M, N \in \Lambda^{\mathsf{let}} \,\}
\end{aligned}$$

Jeder Reduktionsbegriff $\mathbf{R}$ auf Λ^{let} induziert binäre Relationen $\to_R$ (Ein-Schritt R-Reduktion), $\twoheadrightarrow_R$ (R-Reduktion) und $=_R$ (R-Konvertierbarkeit); wobei die Relation $=_R$ der symmetrische Abschluß von $\twoheadrightarrow_R$, und $\twoheadrightarrow_R$ der reflexive und transitive Ab-

schluß von $\rightarrow_R$ ist. Ein-Schritt R-Reduktion ist induktiv wie folgt definiert:

$$
\begin{array}{lll}
(1) & (M,N) \in \mathbf{R} & \Rightarrow M \rightarrow_R N \\
(2) & M \rightarrow_R N & \Rightarrow ZM \rightarrow_R ZN \\
(3) & M \rightarrow_R N & \Rightarrow MZ \rightarrow_R NZ \\
(4) & M \rightarrow_R N & \Rightarrow \lambda x.M \rightarrow_R \lambda x.N \\
(5) & M \rightarrow_R N & \Rightarrow \textbf{let } x = M \textbf{ in } M' \rightarrow_R \textbf{let } x = N \textbf{ in } M' \\
(6) & M \rightarrow_R N & \Rightarrow \textbf{let } x = M' \textbf{ in } M \rightarrow_R \textbf{let } x = M' \textbf{ in } N
\end{array}
$$

Definition 2.23. Ein Reduktionsbegriff $\mathbf{R}$ und ein Typsystem $\vdash$ heißen konsistent, falls für alle $(M,N) \in \mathbf{R}$ gilt:

$$\Gamma \vdash M : \tau \Rightarrow \Gamma \vdash N : \tau$$

Lemma 2.24. *$\beta\eta let$ und $\vdash^{DM}$ sind konsistent.*

Beweis. Durch Induktion über M zeigt man, daß aus der Gültigkeit der Typaussage $\Gamma \vdash (\lambda x.M)N : \tau$ auch die Gültigkeit von $\Gamma \vdash M[N/x] : \tau$ folgt. Ebenso für $\Gamma \vdash \textbf{let } x = M \textbf{ in } N : \tau$ und $\Gamma \vdash N[M/x] : \tau$. Für $\Gamma \vdash (\lambda x.Mx) : \tau$ und $\Gamma \vdash M : \tau$ folgt das Resultat direkt aus den Typisierungsregeln [ABS] und [APP]. □

Satz 2.25 (Subjekt-Reduktion). *Aus $\Gamma \vdash M : \tau$ folgt $\Gamma \vdash M' : \tau$ für alle M' mit $M \twoheadrightarrow_{\beta\eta let} M'$.*

Für keinen der obigen Reduktionsbegriffe sind $=_R$ und $\vdash^{DM}$ konsistent. Es gilt jedoch:

Lemma 2.26 (Typisierungen sind abgeschlossen unter α-Konversion). *Sei M ein Ausdruck und $y \notin \mathcal{FV}(M)$. Dann gilt:*

$$
\begin{array}{rcl}
\Gamma \vdash \lambda x.M : \tau & \iff & \Gamma \vdash \lambda y.M[y/x] \\
\Gamma \vdash \textbf{let } x = N \textbf{ in } M : \tau & \iff & \Gamma \vdash \textbf{let } y = N \textbf{ in } M[y/x] : \tau
\end{array}
$$

Beweis. Durch Induktion über M. (Siehe Proposition 3.1.11 in [Bar92]). □

2.2.3 Typinferenz

Satz 2.27 (Damas-Milner). *Sei Γ eine Typannahme und Γ' eine Instanz von Γ. Falls $\Gamma' \vdash M : \tau'$ eine gültige Typisierung ist, dann gibt es eine allgemeinste Typisierung $S\Gamma \vdash M : \tau$, sodaß eine Substitution R existiert mit $\Gamma' = RS\Gamma$ und $\tau' = R\tau$.*[13] *Darüber hinaus gibt es einen Algorithmus (Algorithmus $\mathcal{W}$) zur Berechnung von $S\Gamma \vdash M : \tau$.*

Algorithmus 2.3. ($\mathcal{W}$) Typinferenz für parametrischen Polymorphismus

---

$\mathcal{W}[\![x]\!]\Gamma = (\emptyset, \tau')$
falls $\Gamma(x) = \forall\alpha_i.\tau$ und $\tau' = \{\alpha_i \mapsto \beta_i\}\tau$,
wobei die β_i „neue“, bisher noch nicht verwendete Typvariablen sind.

$\mathcal{W}[\![M_1\, M_2]\!]\Gamma = (V \circ S_2 \circ S_1, V\beta)$
Falls Substitutionen S_1, S_2, V und Typen τ_1, τ_2 existieren, sodaß
$(S_1, \tau_1) = \mathcal{W}[\![M_1]\!]\Gamma$
$(S_2, \tau_2) = \mathcal{W}[\![M_2]\!]S_1\Gamma$
$V = \mathcal{U}(S_2\tau_1, \tau_2 \to \beta)$
und β eine „neue“ Typvariable ist.

$\mathcal{W}[\![\lambda x.M]\!]\Gamma = (S, S\beta \to \tau)$
Falls eine Substitution S existiert mit
$(S, \tau) = \mathcal{W}[\![M]\!](\Gamma + [x : \beta])$ und β eine „neue“ Typvariable ist.

$\mathcal{W}[\![\textbf{let } x = M_1 \textbf{ in } M_2]\!]\Gamma = (S_2 \circ S_1, \tau_2)$
Falls Substitutionen S_1, S_2 und Typen τ_1, τ_2 existieren, sodaß
$(S_1, \tau_1) = \mathcal{W}[\![M_1]\!]\Gamma$ und
$(S_2, \tau_2) = \mathcal{W}[\![M_2]\!](S_1\Gamma + [x : \mathrm{gen}(S_1\Gamma, \tau_1)])$

In allen anderen Fällen schlägt der Algorithmus fehl.

Der Beweis erfolgt in 2 Teilen: zum einen zeigt man, daß bei Termination von $\mathcal{W}$ eine

[13] Aus Proposition 2.19(iii) und $S\tau = \tau$ folgt dann auch $\mathrm{gen}(\Gamma', \tau') \prec R(\mathrm{gen}(S\Gamma, \tau))$.

korrektes Ergebnis berechnet wird, zum anderen zeigt man, daß $\mathcal{W}$ für typisierbare Ausdrücke terminiert und eine allgemeinste Typisierung berechnet.

Lemma 2.28 ($\mathcal{W}$ ist wohldefiniert). *Falls $\mathcal{W}[\![M]\!]\Gamma$ als Ergebnis (S, τ) liefert, dann ist $S\Gamma \vdash M : \tau$ eine gültige Typisierung.*

Beweis. (Induktion über M)

$\boxed{M \equiv x}$ $S = \emptyset$ und $\tau = \{\alpha_i \mapsto \beta_i\}\tau'$, wobei $\Gamma(x) = \forall\alpha_i.\tau'$. Offensichtlich ist τ eine generische Instanz von $\Gamma(x)$ und daher ist $S\Gamma \vdash x : \tau$ eine gültige Typisierung.

$\boxed{M \equiv M_1\, M_2}$ Nach Definition von $\mathcal{W}$ existieren Substitutionen S_1, S_2, V und eine Typvariable β, sodaß $(S_1, \tau_1) = \mathcal{W}[\![M_1]\!]\Gamma$, $(S_2, \tau_2) = \mathcal{W}[\![M_2]\!]\Gamma$ und $V = \mathcal{U}(S_2\tau_1, \tau_2 \to \beta)$. Nach Induktionshypothese existieren gültige Typisierungen $S_1\Gamma \vdash M_1 : \tau_1$ und $(S_2 \circ S_1)\Gamma \vdash M_2 : \tau_2$. Aus der Abgeschlossenheit unter Substitutionen folgt, daß auch $(V \circ S_2 \circ S_1)\Gamma \vdash M_1 : V(\tau_2 \to \beta)$ und $(V \circ S_2 \circ S_1)\Gamma \vdash M_2 : V\tau_2$ gültige Typisierungen sind. Daher ist auch $(V \circ S_2 \circ S_1)\Gamma \vdash M_1 M_2 : V\beta$ eine gültige Typisierung.

$\boxed{M \equiv \lambda x.N}$ Es existieren S, τ sodaß $(S, \tau) = \mathcal{W}[\![N]\!](\Gamma + [x : \beta])$. Nach Induktionsannahme ist $S(\Gamma + [x : \beta]) \vdash N : \tau$ eine gültige Typisierung. Also folgt aus der [ABS]-Regel, daß auch $S\Gamma \vdash \lambda x.N : S\beta \to \tau$ wohltypisiert ist.

$\boxed{M \equiv \textbf{let } x = M_1 \textbf{ in } M_2}$ Nach Definition von $\mathcal{W}$ existieren Substitutionen S_1, S_2 und Typausdrücke τ_1, τ_2 mit $(S_1, \tau_1) = \mathcal{W}[\![M_1]\!]\Gamma$ und $(S_2, \tau_2) = \mathcal{W}[\![M_2]\!](S_1\Gamma + [x : \mathrm{gen}(S_1\Gamma, \tau_1)])$, sodaß $\Gamma \vdash M : \tau$ und $S_2(S_1\Gamma + [x : \mathrm{gen}(S_1\Gamma, \tau_1)]) \vdash M_2 : \tau_2$ gültige Typisierungen sind. Aufgrund von Proposition 2.20 ist auch $S_2S_1\Gamma \vdash M_1 : S_2\tau_1$ gültig. Da außerdem S_2 keine in $\mathrm{gen}(S_1\Gamma, \tau_1)$ gebundenen Typvariablen involvieren kann, gilt $S_2(\mathrm{gen}(S_1\Gamma, \tau_1)) = (S_2 \circ S_1)(\mathrm{gen}(\Gamma, S_2\tau_1))$. Da $S_1(\tau_1) = \tau_1$ ist auch $(S_2 \circ S_1)\Gamma \vdash \textbf{let } x = M_1 \textbf{ in } M_2 : \tau_2$ eine gültige Typisierung. □

Lemma 2.29 ($\mathcal{W}$ ist vollständig). *Falls $\mathcal{W}[\![M]\!]\Gamma$ als Ergebnis (S, τ) liefert, dann existiert für jede andere gültige Typisierung $\Gamma' \vdash M : \tau'$ mit $\Gamma' \leq \Gamma$ eine Substitution R mit $\Gamma' = RS\Gamma$ und $\tau' = R\tau$.*

Beweis. (Induktion über M)

$\boxed{M \equiv x}$ Dann ist $x \in dom(\Gamma)$, da $\Gamma' \leq \Gamma$ und $\Gamma' \vdash x : \tau'$ eine gültige Typisierung ist, wobei $\Gamma'(x) = \sigma' = \forall\overline{\alpha_n}.\tau'_x$ und $\tau' \prec \sigma'$. Mit der Substitution $S' = \{\alpha_i \mapsto \tau_i\}$

gilt $\tau' = S'\tau'_x$, wobei o.B.d.A. $inv(S') \cap \{\overline{\alpha_n}\} = \emptyset$. Dann terminiert $\mathcal{W}[\![x]\!]\Gamma$ mit dem Ergebnis $S = \emptyset$, $\tau = R_0\tau_x$, wobei $\Gamma(x) = \forall\overline{\alpha_n}.\tau_x$, $R_0 = \{\alpha_i \mapsto \beta_i\}$ für n neue Typvariablen. Nach Voraussetzung gilt $\tau'_x = Q\tau_x$. Man wähle $R = R_0 \circ Q$, dann gilt $\tau' = S'\tau_x = R\tau$ und da $dom(R_0) \cap inv(Q) = \emptyset$ gilt auch $\Gamma' = (R_0 \circ Q)\Gamma = R\Gamma$.

$\boxed{M \equiv M_1\, M_2}$ Dann existieren gültige Typaussagen $\Gamma' \vdash M_1\, M_2 : \tau'_2$, $\Gamma' \vdash M_1 : \tau'_1 \to \tau'_2$ und $\Gamma' \vdash M_2 : \tau'_1$. Nach Induktionsvoraussetzung terminiert $\mathcal{W}[\![M_1]\!]\Gamma$ mit dem Ergebnis (S_1, τ_1) und es existiert eine Substitution R_1 mit $\Gamma' = R_1(S_1\Gamma)$ sowie $\tau'_1 \to \tau'_2 = R_1\tau_1$. Also gilt $\Gamma' \leq S_1\Gamma$ und somit terminiert nach Induktionsannahme auch der Aufruf $\mathcal{W}[\![M_2]\!](S_1\Gamma)$ mit dem Ergebnis (S_2, τ_2) und es existiert eine Substitution R_2, sodaß $\Gamma' = R_2(S_2(S_1\Gamma))$ und $\tau_1 = R_2\tau_2$. Wg. $R_1 = R_2 \circ S_1$ erfüllen τ_1, τ_2 die Bedingungen $\tau'_1 \to \tau'_2 \leq R_2(S_2\tau_1)$ und $\tau'_1 \to \tau'_2 \leq R_2(\tau_2 \to \beta)$, für eine neue Typvariable β, d.h. $R_2(S_2\tau_1)$ und $R_2(\tau_2 \to \beta)$ haben eine gemeinsame untere Schranke in $T_F(\mathcal{V})$. Daher existiert der allgemeinste Unifikator $V = \mathcal{U}(S_2\tau_1, \tau_2 \to \beta)$ und eine Substitution Q mit der Eigenschaft $\Gamma' = Q(V(S_2(S_1\Gamma)))$ und $\tau'_1 = Q(V(S_2(S_1\beta)))$, wie verlangt.

$\boxed{M \equiv \lambda x.N}$ Es existieren Typisierungen $\Gamma' \vdash \lambda x.N : \tau'_1 \to \tau'_2$ und $\Gamma' + [x : \tau'_1] \vdash \tau'_2$ und $\Gamma' \leq \Gamma$. Dann terminiert nach Induktionsannahme $\mathcal{W}[\![N]\!](\Gamma + [x : \beta])$ mit dem Ergebnis (S, τ_r), sodaß eine Substitution R' existiert, für die $\Gamma' + [x : \tau'_1] = R'(S_1(\Gamma + [x : \beta]))$ und $\tau'_2 = R'\tau_r$ gilt. Daraus folgt sofort, daß $\Gamma' = R'(S(\Gamma))$ und $\tau'_1 = R'(S(\beta))$. Nun gilt $\mathcal{W}[\![M]\!]\Gamma = (S, S\beta \to \tau_2)$. Mit der Wahl $R = R' \circ S$ folgt auch $\tau'_1 \to \tau'_2 = R(\beta \to \tau_r)$ und damit die Behauptung.

$\boxed{M \equiv \textbf{let}\ x = M_1\ \textbf{in}\ M_2}$ Nach Voraussetzung existiert eine gültige Typisierung $\Gamma' \vdash$ **let** $x = M_1$ **in** $M_2 : \tau_2$ und damit nach Definition von $\vdash$ Typisierungen $\Gamma' \vdash M_1 : \tau'_1$ und $\Gamma' + [x : \mathrm{gen}(\Gamma', \tau'_1)] \vdash M_2 : \tau'_2$. Nach Induktionsannahme terminiert $\mathcal{W}[\![M_1]\!]\Gamma$ mit dem Ergebnis (S_1, τ_1) und es existiert eine Substitution R_1 mit $\Gamma' = R_1(S_1\Gamma)$, sowie $\tau'_1 = R_1\tau_1$. Betrachte $\sigma = \mathrm{gen}(\Gamma', \tau'_1) = \mathrm{gen}(R_1(S_1\Gamma), R_1\tau_1)$. Aufgrund von Proposition 2.19(iii) gilt $\sigma \prec R_1(\mathrm{gen}(S_1\Gamma, \tau_1))$ und somit folgt Aufgrund von Proposition 2.21, daß mit $\Gamma'' = \Gamma' + [x : R_1(\mathrm{gen}(S_1\Gamma, \tau_1))]$ auch $\Gamma'' \vdash M_2 : \tau'_2$ eine gültige Typisierung ist. $\Gamma'' = R_1(\Gamma + [x : \mathrm{gen}(S_1\Gamma, \tau_1)])$ erfüllt nun die Vorbedingung zur Anwendung der Induktionsannahme, sodaß $\mathcal{W}[\![M_2]\!](\Gamma + [x : \mathrm{gen}(S_1\Gamma, \tau_1)])$ terminiert und als Ergebnis (S_2, τ_2) liefert, wobei $\Gamma'' = R_2(S_2(S_1(\Gamma + [x : \mathrm{gen}(S_1\Gamma, \tau_1)]))$ und $\tau'_2 = R_2\tau_2$. Daraus folgt auch $\Gamma' = R_2(S_2(S_1(\Gamma)))$ und damit die Behauptung. □

Algorithmus $\mathcal{W}$ bildet auch die Grundlage der Implementierung des Typinferenzalgo-

rithmus für das SAMPλE-System ohne Konversionen. Die SAMPλE-Implementierung verwendet allerdings einen modifizierten Unifikationsalgorithmus, der die verbreitete DAG-Repräsentationstechnik von Typausdrücken verwendet und eine Implementierung der Operation $\text{gen}(\Gamma, \tau)$ erlaubt, die linear in Größe des zu abstrahierenden Typausdrucks τ ist, unabhängig von der Größe der Typannahme Γ.[14]

2.2.4 Komplexitätsresultate

Aufgrund der Möglichkeit, Unifikation mit linearem Aufwand implementieren zu können, des somit linearen Aufwands für die Typisierung let-freier Ausdrücke, sowie der Tatsache, daß im langjährigen, praktischen Einsatz von ML, bzw. Standard-ML, Hope und Miranda und auch SAMPλE, keine ungewöhnlichen Laufzeiten der Typinferenz auffielen, wurde lange Zeit angenommen, daß der Algorithmus $\mathcal{W}$ polynomial zeitbeschränkt ist. Dies ist jedoch nicht der Fall: zuerst zeigten Mitchell und Kanellakis in [KM89], daß das Typisierungsproblem für let-Polymorphismus NP-schwer ist. Dieses Resultat wurde später von Mairson in [Mai90] verbessert: Typisierbarkeit im ML-Stil ist deterministisch, Exponential-Zeit vollständig (DEXPTIME).

Zwar kann durch die Darstellung von Typausdrücken durch gerichtete Graphen eine exponentielle Reduktion des Platzbedarfs gegenüber einer Baumdarstellung der Terme erzielt werden, der Kern des Problems liegt jedoch in der Möglichkeit, mit Hilfe des let-Konstrukts Typausdrücke zu erzeugen, die exponentiell viele Typvariablen enthalten, wie das folgende Beispiel zeigt:

$$
\begin{array}{l}
\textbf{let } x_0 \equiv \lambda x.\lambda y.\lambda z.z\ x\ y \\
\textbf{in let } x_1 \equiv \lambda y.x_0\ y\ y \\
\quad \textbf{in let } x_2 \equiv \lambda y.x_1\ (x_1\ y) \\
\quad\quad \textbf{in let } x_3 \equiv \lambda y.x_2\ (x_2\ y) \\
\quad\quad\quad \textbf{in let } x_4 \equiv \lambda y.x_3\ (x_3\ y) \\
\quad\quad\quad\quad \textbf{in let } x_5 \equiv \lambda y.x_4\ (x_4\ y) \\
\quad\quad\quad\quad\quad \textbf{in } x_5\ (\lambda z.z)
\end{array}
$$

Der Ausdruck $x_0\, a\, b$ kodiert die Paarbildung (a, b) im reinen Lambda-Kalkül. x_0 hat den allgemeinsten Typ $\forall\alpha, \beta, \gamma.\alpha \to \beta \to (\alpha \to \beta \to \gamma) \to \gamma$. x_1 hat den

[14] Eine einfache Modula2-Implementierung von $\mathcal{W}$ wird in [Car87] beschrieben.

allgemeinsten Typ $\forall\alpha,\gamma.\alpha \to (\alpha \to \alpha \to \gamma) \to \gamma$. x_2 hat den Typ $\forall\alpha,\beta,\gamma.a \to (((\alpha \to \alpha \to \beta) \to \beta) \to ((\alpha \to \alpha \to \beta) \to \beta) \to \gamma) \to \gamma$. Der Typ für x_3 lautet $\forall\alpha,\beta,\gamma,\delta,\epsilon.\alpha \to (((((((\alpha \to \alpha \to \beta) \to \beta) \to ((\alpha \to \alpha \to \beta) \to \beta) \to \gamma) \to \gamma) \to ((((\alpha \to \alpha \to \beta) \to \beta) \to ((\alpha \to \alpha \to \beta) \to \beta) \to \gamma) \to \gamma) \to \delta) \to \delta) \to ((((((\alpha \to \alpha \to \beta) \to \beta) \to ((\alpha \to \alpha \to \beta) \to \beta) \to \gamma) \to \gamma) \to ((((\alpha \to \alpha \to \beta) \to \beta) \to ((\alpha \to \alpha \to \beta) \to \beta) \to \gamma) \to \gamma) \to \delta) \to \delta) \to \epsilon) \to \epsilon$ und die lineare Darstellung des Typs von x_4 füllt schon 2 Seiten.[15]

Angesichts der o.a. Komplexitätsresultate kann man sich natürlich die Frage stellen, ob die Entscheidbarkeit der Typisierbarkeit überhaupt von Relevanz ist. Der Autor ist jedoch der Ansicht, daß man eher die Aussagekraft der *worst case* Abschätzung von Algorithmen für ihren praktischen Einsatz in Frage stellen sollte, da in der Praxis offensichtlich keine für die Typinferenz problematischen Programme geschrieben werden. Betrachtet man nämlich z.B. den Beweis von Mairson, dann sieht man, daß dort die Berechnung von nichtdeterministischen Turingmaschinen im Typsystem kodiert wird. Im praktischen Einsatz käme sicher niemand auf die Idee, das Typsystem für eine solche Aufgabe auszunutzen, sondern würde wohl eher ein Programm schreiben, das die gleiche Aufgabe erfüllt.

Für die Komplexitätsbetrachtungen im Zusammenhang mit den Erweiterungen des Milner-Systems für Überladungen und Konversionen bedeutet das Resultat, daß wir uns auf die Analyse der Komplexität der Typinferenz let-freier Lambda-Terme beschränken werden.

2.2.5 Semantik von Typen und Ausdrücken

Die zweite wichtige Eigenschaft des Damas-Milner-Typsystems ist die Tatsache, daß wohltypisierte Programme garantiert frei von Laufzeittypfehlern sind. Der Beweis basiert auf einer denotationalen Semantikdefinition für die Beispielsprache. Dabei wird durch Induktion über die Definition der Semantikfunktion bewiesen, daß die Bedeutung eines wohltypisierten Programms, also sein denotierbarer Wert, immer

[15] Die Ausdrücke x_2, x_3, etc., sind nur Funktionskompositionen ($x_{i+1} = x_i \circ x_i$). Somit scheint der Grund für das Anwachsen der Typausdrücke hauptsächlich in der Möglichkeit zu liegen, Funktionskomposition polymorpher Funktionen ohne Angabe der beteiligten Funktionstypen notieren zu können. Müßten Funktionstypen angegeben werden, hätte man einen linearen Algorithmus.

ein Element einer bestimmten Teilmenge aller denotierbaren Werte ist. Diese Teilmenge wird allein durch den zugeordneten Typ bestimmt und enthält nie den Wert *wrong*, durch den Laufzeittypfehler repräsentiert werden.[16] Besonders hervorzuheben ist dabei, daß die Semantik völlig unabhängig von der Typisierung definiert wird.

Um die Bedeutung von Ausdrücken beschreiben zu können, postulieren wir die Existenz eines semantischen Bereichs (Domain) **DV** als Lösung (i.e. kleinsten Fixpunkt) einer rekursiven Domaingleichung[17]

$$\mathbf{DV} = \mathbf{W} \oplus S_{f_1}(\underbrace{\mathbf{DV}, \ldots, \mathbf{DV}}_{\rho(f_1)-\mathrm{mal}}) \oplus \cdots \oplus S_{f_n}(\underbrace{\mathbf{DV}, \ldots, \mathbf{DV}}_{\rho(f_n)-\mathrm{mal}})$$

wobei $\oplus$ die disjunkte Vereinigung von Bereichen bezeichnet, und

1. $\mathbf{W} = \{\cdot\}$ der Domain des Wertes $\cdot$, der zur Modellierung von Laufzeitfehlern verwendet wird. Als Element von **DV** wird $\cdot$ durch den Bezeichner *wrong* notiert.

2. S_f ist ein zum Typkonstruktor f korrespondierender *Bereichskonstruktor*, wobei wir zumindest die folgenden Konstruktoren voraussetzen:

S_{int}	- der flache Bereich der ganzen Zahlen $\{\ldots, -1, 0, 1, \ldots\}$
S_{real}	- der flache Bereich der Fließkomma-Zahlen
S_{bool}	- der flache Bereich der Wahrheitswerte $\{t, f\}$
$S_\times(A, B)$	- das kartesische Produkt von Bereichen
$S_\rightarrow(A, B)$	- der Bereich der stetigen Funktionen von A nach B

Wir verwenden folgende Notation[18]:

$\boldsymbol{D}_f$	- der Unterbereich von **DV**, der von S_f erzeugt wird.
$x \sqsubseteq \boldsymbol{D}_f$	- Test ob x in $\boldsymbol{D}_f$ liegt, wobei $\bot \sqsubseteq \boldsymbol{D}_f = \bot_{\mathbf{DV}}$
x in **DV**	- Injektion eines Elementes eines Unterbereichs in **DV**
$x \vert \boldsymbol{D}_f$	- Projektion von $x \in \mathbf{DV}$ auf den Unterbereich $\boldsymbol{D}_f$
$a \rightarrow b, c$	- b, falls $a = t$ in **DV**, c, falls $a = f$ in **DV**, $\bot_{\mathbf{DV}}$ sonst.

[16] Daher die Phrase "well typed programs can't go wrong".
[17] Wir verstehen im Folgenden unter Bereich eine vollständige partielle Ordnung (CPO).
[18] Siehe z.B. [Sto77, Seiten 91–95].

Die Bedeutung eines Ausdrucks M definieren wir über eine semantische Funktion $\mathcal{E}[\![M]\!]$, die endliche Abbildungen $\eta \in \mathit{Env} = \mathbf{Id} \mapsto \mathbf{DV}$ auf denotierbare Werte abbildet:

$$\begin{array}{lll}
\mathcal{E} & : & \Lambda^{\mathsf{let}} \to \mathit{Env} \to \mathbf{DV} \\
\mathcal{E}[\![x]\!]\eta & = & \eta(x) \\
\mathcal{E}[\![\lambda x.M]\!]\eta & = & (\lambda v.\mathcal{E}[\![M]\!]\eta\{v/x\}) \\
\mathcal{E}[\![M_1 M_2]\!]\eta & = & f \in \boldsymbol{D}_{\to} \to (f|\boldsymbol{D}_{\to})\, v,\ \mathit{wrong} \\
 & & \text{wobei}\quad f = \mathcal{E}[\![M_1]\!]\eta,\ v = \mathcal{E}[\![M_2]\!]\eta \\
\mathcal{E}[\![\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2]\!]\eta & = & \mathcal{E}[\![M_1]\!]\eta\{\mathcal{E}[\![M_2]\!]\eta/x\}
\end{array}$$

wobei

$$\eta\{v/x\} \stackrel{\text{def}}{=} \lambda y. \begin{cases} v & \text{falls } x = y, \\ \eta(y) & \text{sonst.} \end{cases}$$

das Überschreiben von η mit x an der Stelle v bezeichnet.[19]

Wie man leicht sieht, gilt $\mathcal{E}[\![\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2]\!] = \mathcal{E}[\![(\lambda x.M_2)M_1]\!]$.

Der einzige aus den Gleichungen unmittelbar ersichtliche Laufzeittypfehler ist die Anwendung eines nichtfunktionalen Wertes auf ein Argument. Wir wollen jedoch annehmen, daß die Anwendung vordefinierter Funktionen auf Werte außerhalb ihres Argumentbereichs auch den Wert *wrong* liefert. Also beispielsweise

$$\eta(\mathit{intadd}) = \lambda a.\lambda b.\ a \in \boldsymbol{D}_{\mathit{int}} \to (b \in \boldsymbol{D}_{\mathit{int}} \to (a +_{\mathit{int}} b)\ \mathsf{in}\ \mathbf{DV}, \mathit{wrong}), \mathit{wrong}$$

Typen interpretieren wir als Elemente in dem Verband der schwachen Ideale $\mathbf{I}(\mathbf{DV})$ (siehe [MS82, MPS84]). Ideale $I \subseteq \mathbf{DV}$ sind nicht leer, enthalten nicht *wrong*, sind nach unten abgeschlossen ($x \in I \wedge y \sqsubseteq x \Rightarrow y \in I$), und abgeschlossen gegenüber Grenzwertbildung, d.h., für jede aufsteigende Sequenz $\{x_i\}$ in $\mathbf{DV}$ gilt $(\forall x_i.\ x_i \in I) \Rightarrow \bigsqcup\{x_i\} \in I$. Darüber hinaus ist $\mathbf{I}(\mathbf{DV})$ auch unter Vereinigung und Schnittmengenbildung abgeschlossen. Für irgendein $f \in F$ mit Stelligkeit n, sei

[19] An dieser Stelle sind wir notationell etwas lax: auf der rechten Seite der Semantikdefinitionsgleichungen bezeichnen wir mit $\lambda x.e(x)$ die Funktion f, die x auf $e(x)$ abbildet. Wir verzichten auf eine syntaktische Kennzeichnung, etwa durch Unterstreichung, da alle syntaktischen Vorkommen durch die denotationalen Klammern $[\![]\!]$ eingeschlossen werden.

$\mathbf{i}(f)(I_1, \ldots, I_n)$ das Bild von $S_f(I_1, \ldots, I_n)$ in **DV**. Beispielsweise

$$\begin{aligned}
&\mathbf{i}(int) = \boldsymbol{D}_{int} \text{ in } \mathbf{DV}\\
&\mathbf{i}(real) = \boldsymbol{D}_{real} \text{ in } \mathbf{DV}\\
&\mathbf{i}(\times)(I, J) = \{(a, b) \mid a \in I, b \in J\} \text{ in } \mathbf{DV}\\
&\mathbf{i}(\rightarrow)(I, J) = \{f \in \boldsymbol{D}_{\rightarrow} \mid x \in I \Rightarrow f(x) \in J\} \text{ in } \mathbf{DV}
\end{aligned}$$

Definition 2.30 (Ideal-Semantik von Typausdrücken). Sei $\varphi \in \mathbf{Tenv} = \mathcal{V} \mapsto \mathbf{I}(\mathbf{DV})$, dann bildet die Funktion $\mathcal{T}$ Typschemata σ auf Ideale ab, falls φ eine Zuweisung von Idealen auf die freien Typvariablen von σ ist:

$$\begin{aligned}
\mathcal{T}[\![\alpha]\!]\varphi &= \varphi(\alpha)\\
\mathcal{T}[\![f(\tau_1, \ldots, \tau_n)]\!]\varphi &= \mathbf{i}(f)(\mathcal{T}[\![\tau_1]\!]\varphi, \ldots, \mathcal{T}[\![\tau_n]\!]\varphi)\\
\mathcal{T}[\![\forall\alpha.\sigma]\!]\varphi &= \textstyle\bigcap_{t \in T_F(\emptyset)} \mathcal{T}[\![\sigma]\!]\varphi\{\mathcal{T}[\![t]\!]\varphi/\alpha\}
\end{aligned}$$

Proposition 2.31 ($\mathcal{T}$ ist wohldefiniert). *Falls $\mathcal{FV}(\sigma) \subseteq dom(\varphi)$ und $\varphi \in \mathcal{V} \rightarrow \mathbf{I}(\mathbf{DV})$ dann ist $\mathcal{T}[\![\sigma]\!]\varphi$ ein Ideal.*

Beweis. Direkt aus der Definition von $\mathcal{T}$. □

Proposition 2.32. *Sei $\sigma = \forall\overline{\alpha_n}.\tau$ ein Typschema und $\mathcal{FV}(\sigma) \subseteq dom(\varphi)$, dann gilt $v \in \mathcal{T}[\![\sigma]\!]\varphi$ genau dann, wenn $v \in \mathcal{T}[\![S\tau]\!]\varphi$ für jede Substitution $S = \{\alpha_i \mapsto t_i\}$, die den gebundenen Typvariablen von σ Monotypen $t \in T_F(\emptyset)$zuordnet.*

Beweis. Direkt aus der Definition von $\mathcal{T}$. □

Proposition 2.33. *Sei v ein denotierbarer Wert, σ ein Typschema und $\varphi \in Tenv$. Falls $v \in \mathcal{T}[\![\sigma]\!]\varphi$ dann ist $v \in \mathcal{T}[\![\sigma']\!]\varphi$ für jede generische Instanz $\sigma' \prec \sigma$, die $\mathcal{FV}(\sigma') \subseteq dom(\varphi)$ erfüllt.*

Beweis. Aus der Definition von $\mathcal{T}$ und der Tatsache, daß generische Instanzierung nur gebundene Typvariablen verändert. □

Proposition 2.34. *Falls $f \in \mathcal{T}[\![\forall\overline{\alpha_k}.\tau \rightarrow \tau']\!]\varphi$, $v \in \mathcal{T}[\![\forall\overline{\beta_m}.\tau]\!]\varphi$ und $\overline{\beta_m} \subseteq \overline{\alpha_k}$ dann gilt $(f|F)v \in \mathcal{T}[\![\forall\overline{\gamma_n}.\tau']\!]\varphi$, wobei $\overline{\gamma_n} = \mathcal{V}(\tau) - \mathcal{FV}(\forall\overline{\alpha_k}.\tau \rightarrow \tau')$.*

Beweis. Nach Definition von $\mathcal{T}$ und $\mathbf{i}(\rightarrow)$. □

Satz 2.35 (Typinferenz verhindert Typfehler zur Laufzeit). *Ist* $\Gamma \vdash M : \tau$ *eine gültige Typisierung und* $x : \sigma \in \Gamma \Rightarrow \eta(x) \in \mathcal{T}[\![\sigma]\!]\varphi$*, wobei* $\mathcal{FV}(M) \subseteq dom(\eta)$ *und* $\mathcal{FV}(\Gamma) \cup \mathcal{FV}(\mathrm{gen}(\Gamma, \tau)) \subseteq dom(\varphi)$*, dann gilt* $\mathcal{E}[\![M]\!]\eta \in \mathcal{T}[\![\mathrm{gen}(\Gamma, \tau)]\!]\varphi$*.*

Beweis. (Induktion über M)

$\boxed{M \equiv x}$ Es existiert eine gültige Typisierung $\Gamma + [x : \sigma] \vdash x : \tau$ wobei τ eine generische Instanz von σ ist. Da $\mathcal{E}[\![x]\!]\eta = \eta(x) \in \mathcal{T}[\![\sigma]\!]\varphi$ und $\mathrm{gen}(\Gamma, \tau)$ eine generische Instanz von σ ist, folgt aus Proposition 2.33: $\eta(x) \in \mathcal{T}[\![\mathrm{gen}(\Gamma, \tau)]\!]\varphi$.

$\boxed{M \equiv M_1\, M_2}$ Es existieren Typisierungen $\Gamma \vdash M_1 M_2 : \tau$, $\Gamma \vdash M_1 : \tau' \to \tau$ und $\Gamma \vdash M_2 : \tau'$, sodaß $f \in \mathcal{T}[\![\mathrm{gen}(\Gamma, \tau' \to \tau)]\!]\varphi$ und $v \in \mathcal{T}[\![\mathrm{gen}(\Gamma, \tau')]\!]\varphi$, wobei $f = \mathcal{E}[\![M_1]\!]\eta, v = \mathcal{E}[\![M_2]\!]\eta$. Nach Proposition 2.34 folgt $(f|F)v \in \mathcal{T}[\![\mathrm{gen}(\Gamma, \tau)]\!]\varphi$.

$\boxed{M \equiv \lambda x.N}$ Es existieren Typisierungen $\Gamma \vdash \lambda x.N : \tau' \to \tau$ und $\Gamma + [x : \tau'] \vdash N : \tau$ und wir müssen zeigen, daß $f \in \mathcal{T}[\![\mathrm{gen}(\Gamma, \tau' \to \tau)]\!]\varphi$, wobei $f = \lambda v.\mathcal{E}[\![N]\!]\eta\{v/x\}$. Aufgrund von Lemma 2.32 ist dies äquivalent zu $f \in \mathcal{T}[\![S(\tau' \to \tau)]\!]\varphi$ für jede Substitution S die den in $\mathrm{gen}(\Gamma, \tau' \to \tau)$ gebundenen Typvariablen monomorphe Typen zuordnet. Sei S eine solche Substitution: Nach Proposition 2.20 ist $\Gamma + [x : S\tau'] \vdash N : S\tau$ wohltypisiert und wir können daher die Induktionshypothese anwenden und erhalten: $\mathcal{E}[\![N]\!]\eta\{v/x\} \in \mathcal{T}[\![S\tau]\!]\varphi$, genau dann, wenn $v \in \mathcal{T}[\![S\tau']\!]\varphi$. Damit ist aber auch, wie verlangt, $f \in \mathcal{T}[\![S(\tau' \to \tau)]\!]\varphi$.

$\boxed{M \equiv \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2}$ Es existieren gültige Typisierungen $\Gamma \vdash M_1 : \tau_1$, $\Gamma \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau_2$, und $\Gamma' \vdash M_2 : \tau_2$, sodaß $\mathcal{E}[\![M_1]\!]\eta \in \mathcal{T}[\![\mathrm{gen}(\Gamma, \tau_1)]\!]\varphi$, wobei $\Gamma' = \Gamma + [x : \mathrm{gen}(\Gamma, \tau)]$. Sei $\eta' = \eta\{\mathcal{E}[\![M_1]\!]\eta/x\}$. Es ist leicht zu verifizieren, daß $\Gamma'(y) = \sigma \Rightarrow \eta'(y) \in \mathcal{T}[\![\sigma]\!]\varphi$ und (nach I.H.) $\mathcal{E}[\![M_2]\!]\eta' \in \mathcal{T}[\![\mathrm{gen}(\Gamma', \tau_2)]\!]\varphi$. Nun ist aber $\mathrm{gen}(\Gamma, \tau_2) = \mathrm{gen}(\Gamma', \tau_2)$, da $\mathcal{FV}(\Gamma) = \mathcal{FV}(\Gamma')$, was auch $\mathcal{E}[\![\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2]\!]\eta \in \mathrm{gen}(\Gamma, \tau_2)$ impliziert. □

Kapitel 3

Parametrische Überladungen

Traditionelle Programmiersprachen unterscheiden sich von Typinferenz-basierten Programmiersprachen durch die Tatsache, daß nur Basisoperatoren überladen sind und polymorphe Typen besitzen können und der Typ aller vom Programmierer deklarierten Bezeichner monomorph ist. Auf der Ebene von Typdeduktionssystemen kann dies dargestellt werden, indem man von untypisierten Parametern übergeht zu Parametern mit Typdeklarationen und zusätzlich erlaubt, daß vordefinierten Bezeichnern eine Menge von Typen zugeordnet werden kann. Ein entsprechendes Typdeduktionssystem ist in Abbildung 3.1 angegeben.

Üblicherweise fordert man dabei, daß sämtliche Überladungen zur Übersetzungszeit aufgelöst werden können, d.h. bei einer gegebenen Basistypannahme Γ_0 muß die Menge der inferierbaren Typen immer einelementig sein: $|\{t \mid \Gamma_0 \vdash^t M : t\}| = 1$. Wie man sich leicht überlegt, müssen dazu zumindest die Typen der Überladungsinstanzen jedes überladenen Operators *paarweise nicht unifizierbar* sein.

Darüber hinaus kann man 2 Klassen von Programmiersprachen unterscheiden: solche die *kontextunabhängige* und solche die *kontextabhängige* Überladungen unterstützen. Kontextunabhängige Überladungen sind dadurch gekennzeichnet, daß der Typ der Argumente in einer Funktionsapplikation ausreicht, die beabsichtigte Überladungsinstanz und somit auch den Resultattyp der Applikation zu bestimmen. Bei kontextabhängigen Überladungen wird dagegen auch der Resultattyp zur Auflösung von Überladungen herangezogen.

Die meisten der bekannteren Programmiersprachen unterstützen nur kontextunabhängige Überladungen: die rein prozeduralen Sprachen PASCAL [JW78], MODULA2 [Wir83] und C [KR88], sowie die objektorientierten Sprachen C++ [ES92] und Java [GJSB00]. Von den bekannteren Sprachen unterstützt lediglich ADA [als83] kontextabhängige Überladungen, auch für benutzerdefinierte Bezeichner.

$$\text{[VAR]} \quad \frac{t \prec \sigma \ , \ \sigma \in \Gamma(x)}{\Gamma \vdash^t x : t}$$

$$\text{[ABS]} \quad \frac{\Gamma + [\, x : \{t\} \,] \vdash^t M : t'}{\Gamma \vdash^t \lambda x : t . M : t \to t'}$$

$$\text{[APP]} \quad \frac{\Gamma \vdash^t M_1 : t \to t' \ , \ \Gamma \vdash^t M_2 : t}{\Gamma \vdash^t M_1 M_2 : t'}$$

$$\text{[LET]} \quad \frac{\Gamma \vdash^t M_1 : t \ , \ \Gamma + [\, x : \{t\} \,] \vdash^t M_2 : t'}{\Gamma \vdash^t \textbf{let } x : t = M_1 \textbf{ in } M_2 : t'}$$

Abbildung 3.1: Typdeduktion für Überladungen in traditionellen Programmiersprachen

Für nicht polymorphe, streng typisierte Programmiersprachen mit kontextunabhängigen Überladungen, lassen sich Überladungen immer „bottom up“ durch den Vergleich der Argumenttypen mit allen Überladungsinstanzen eines Operators auflösen, da unterversorgte Applikationen aufgrund fehlender Unterstützung für Funktionen höherer Ordnung in diesen Programmiersprachen nicht auftreten.

Für kontextabhängige Überladungen kann man entweder einen 2 phasigen „bottom up, top down“ Algorithmus verwenden [GR80], oder einen einphasigen „bottom up“ Algorithmus mit einer intelligenten Graph-Datenstruktur zur Darstellung der Menge von möglichen Typen [Bak82]. In beiden Fällen erhält man polynomial zeitbeschränkte Typisierungsalgorithmen.

Für Sprachen mit parametrisch polymorphen Typsystemen stellt sich die Situation jedoch ungleich komplexer dar: durch das Fehlen der Typdeklarationen für Funktionsparameter, müssen sämtliche Überladungsinstanzen getestet werden und man erhält für let-freie Terme einen NP-vollständigen Typinferenzalgorithmus [BMS80]. Ein auf einer endlichen Menge paarweise nicht unifizierbarer Überladungsinstanzen basierendes Typdeduktionssystem ist in Abbildung 3.2 angegeben.

Daß das Problem in NP liegt, ist klar: durch Raten der benötigten Überladungsin-

$$\text{[VAR]} \quad \frac{\tau \prec \sigma \ , \ \sigma \in \Gamma(x)}{\Gamma \vdash x : \tau}$$

$$\text{[ABS]} \quad \frac{\Gamma + [\, x : \{\tau\}\,] \vdash M : \tau'}{\Gamma \vdash \lambda x.M : \tau \to \tau'}$$

$$\text{[APP]} \quad \frac{\Gamma \vdash M_1 : \tau \to \tau' \ , \ \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 M_2 : \tau'}$$

$$\text{[LET]} \quad \frac{\Gamma \vdash M_1 : \tau \ , \ \Gamma + [\, x : \{\mathrm{gen}(\Gamma, \tau)\}\,] \vdash M_2 : \tau'}{\Gamma \vdash \textbf{let } x = M_1 \textbf{ in } M_2 : \tau'}$$

Abbildung 3.2: Typdeduktion für endliche Überladungsinstanzmengen

stanzen reduziert sich das Problem auf einfache let-freie Typinferenz, die bekanntermaßen mit linearem Aufwand implementiert werden kann. Die NP-Vollständigkeit folgt durch Reduktion der Erfüllbarkeit boolescher Ausdrücke (SAT, siehe z.B. [AHU74, Kapitel 10]) auf überladene Typinferenz: unter der Basistypannahme

$$\begin{aligned}\Gamma_0 = [\ & a : 0 \to 0 \to 0,\ a : 0 \to 1 \to 0,\ a : 1 \to 0 \to 0,\ a : 1 \to 1 \to 1 \\ & o : 0 \to 0 \to 0,\ o : 0 \to 1 \to 1,\ o : 1 \to 0 \to 1,\ o : 1 \to 1 \to 1 \\ & n : 0 \to 1,\ n : 1 \to 0\ ]\end{aligned}$$

hat der dem booleschen Ausdruck b über den Variablen $x_1, \ldots, x_k$ zugeordnete Term $\lambda x_1. \cdots \lambda x_k.\widehat{b}$ genau dann einen Typ $t \in \{0, 1\}$, falls b erfüllbar ist. Dabei ist $\widehat{b}$ wie folgt definiert:

$$\begin{aligned}\widehat{x_i} &= x_i \\ \widehat{\neg b} &= (n\ \widehat{b}) \\ \widehat{b_1 \wedge b_2} &= (a\ \widehat{b_1}\ \widehat{b_2}) \\ \widehat{b_1 \vee b_2} &= (o\ \widehat{b_1}\ \widehat{b_2})\end{aligned}$$

Abgesehen von diesem Komplexitätsproblem, das aus ähnlichen Gründen wie die DEXPTIME-Vollständigkeit des „einfachen“ parametrischen Polymorphismus in der

Praxis kaum zu Problemen führt, hat das Typdeduktionssystem für endliche Überladungsinstanzmengen aber zwei weitere, nach Ansicht des Autors entscheidendere, Nachteile:

Zwar können vordefinierte Operatoren überladen verwendet werden, es ist jedoch nicht möglich, let-eingeführte Bezeichner überladen zu verwenden. Dies folgt aus der Tatsache, daß zur Deduktion des Typs eines Ausdrucks der Form **let** $x = M_1$ **in** M_2, zur Typisierung des Unterausdruck M_2, dem Bezeichner x eine einelementige Menge möglicher Typen zugeordnet wird. Somit kann x zwar mehrere Typen annehmen, jedoch können zur Typisierung von M_2 nur *polymorphe Instanzen* dieses einzigen Typs für eine Deduktion verwendet werden. Dennoch wurde diese Vorgehensweise mehrfach in polymorphen Programmiersprachen verwendet: in HOPE (siehe [BMS80]) für alle überladene Operatoren, in Standard-ML für die arithmetischen Operatoren (siehe z.B. [Har86, Seiten 24–26]).

Der zweite Nachteil der Vorgehensweise ist die fehlende Möglichkeit, einen polymorphen Gleichheitsoperator korrekt typisieren zu können. Im einfachsten Fall soll dabei polymorphe Gleichheit auf allen Datentypen definiert sein, außer auf Funktionen. Diese Restriktion läßt sich mit endlichen Typmengen offensichtlich nicht mehr ausdrücken. Man kann jedoch eine induktive Definition für die Menge der zulässigen monomorphen Typinstanzen angeben: ein Argumenttyp ist für den Gleichheitsoperator zulässig, wenn er in der kleinsten Menge Z liegt, die folgende Bedingung erfüllt:

$$f \in F_n \wedge f \neq (\rightarrow) \wedge \forall i = 1..n : t_i \in Z \Longrightarrow f(t_1, \ldots, t_n) \in Z$$

Es gibt aber keine endliche Menge von Typausdrücken I, die Z als Instanzmenge generiert: falls $\tau \in I$ mit $\mathcal{V}(\tau) \neq \emptyset$, dann ist natürlich $\{\alpha \mapsto (b \rightarrow b)\}\tau$ für jedes $b \in F_o$ eine Instanz, die den Funktionstypkonstruktor enthält, und damit nicht in Z liegt. Um solche Instanzen auszuschließen, muß eine Repräsentation der Menge Z durch einen einzigen Typausdruck gefunden werden. In Standard-ML (siehe [Mil84]) wurde dazu der Typvariablenbegriff um die sogenannten Typvariablen für „equality-types“ erweitert: statt $\forall\alpha.\alpha \rightarrow \alpha \rightarrow \alpha$ wird der Typ des Gleichheitsoperators durch $\forall\,'\alpha.'\alpha \rightarrow {}'\alpha \rightarrow {}'\alpha$ beschrieben und gefordert, daß Typvariablen, die mit einem Apostroph versehen sind, nur durch Typausdrücke ersetzt werden dürfen, die weder den Typkonstruktor $\rightarrow$, noch einfache Typvariablen enthalten dürfen.

Im Folgenden zeigen wir, daß sich dieses Konzept der „equality-types“ systematisieren und erweitern läßt[1], sodaß sowohl die Überladungen für arithmetische Operatoren als auch der Gleichheitsoperator und verwandte Operatoren einheitlich behandeln werden können und insbesondere in benutzerdefinierten Funktionen (i.e. let-eingeführten Bezeichnern) vollständig abstrahierbar sind. Überladungen, die sich so behandeln lassen, nennen wir in Anlehnung an den Begriff des parametrischen Polymorphismus *parametrische Überladungen*.

Die restlichen Abschnitte dieses Kapitels sind wie folgt organisiert: Zunächst geben wir eine formale Definition der parametrischen Überladungen auf der Basis von Überladungsschemata und Überladungsannahmen für vordefinierte überladene Operatoren (Abschnitt 3.1). Dann definieren wir eine Typisierbarkeitsrelation (Abschnitt 3.2) sowie die Semantik von Typen und Ausdrücken (Abschnitt 3.3). Anschließend präsentieren wir einen unitären Unifikationsalgorithmus und erhalten als Ergebnis die Existenz allgemeinster Typen für parametrische Überladungen (Abschnitt 3.4). Wir untersuchen die Verbindung zur Unifikation in ordnungssortierten freien Termalgebren (3.4.1) und erweitern die Überladungsunifikation auf Konstruktorvariablen (3.4.2).

Abschließend untersuchen wir die Konsequenzen der Erweiterung der Konzepte auf benutzerdefinierte Überladungen (Abschnitt 3.5), wobei wir verschiedene Semantik-Definitionen untersuchen: Überladungsauflösung zur Laufzeit (Abschnitte 3.5.2 und 3.5.3) sowie Übersetzung nach Mini-ML (Abschnitt 3.5.4). In Abschnitt 3.6 vergleichen wir das Konzept der parametrischen Überladungen mit dem Haskell-Typkonzept. Eine Diskussion der Resultate und verwandter Arbeiten beschließt das Kapitel (Abschnitt 3.7).

3.1 Überladungsschemata u. Überladungsannahmen

Wir verlangen zunächst, daß die Menge der möglichen Typen eines überladenen Operators immer durch ein Paar repräsentiert wird, bestehend aus einem speziellen Typausdruck, genannt *Überladungsschema*, sowie einer Beschreibung der Menge

[1] Dem Autor ist keine Veröffentlichung bekannt, die eine formale Definition der Semantik der „equality-types“ bzw. eines Unifikations- und Typinferenzalgorithmus enthält.

der möglichen Instanzen, die aus dem Überladungsschema gebildet werden können. Beides zusammen nennen wir *Überladungsannahme*.

Definition 3.1 (Überladungsschema). Sei \$ ein spezielles Symbol, sodaß $\$ \notin F \cup \mathcal{V}$, dann nennen wir ω ein Überladungsschema, falls $\omega = \omega_1 \to \ldots \to \omega_n$ und für alle $i \in 1..n$, ist ω_i entweder ein Typausdruck τ, oder \$, und $\omega_n = \$ \Rightarrow \exists i \in 1..(n-1)$ sodaß $\omega_i = \$$.

Das spezielle Überladungssymbol \$ markiert die Argumentpositionen eines überladenen Operators, die mit unterschiedlichen Typen instanziert werden dürfen.

Typische Beispiele für Überladungsschemata sind:

$\$ \to \mathit{int}$	ein diskretes Maß
$\$ \to \mathit{real}$	ein stetiges Maß
$\$ \to \$$	unäre Operatoren: $\mathrm{succ}, \mathrm{pred}, \mathrm{abs}$
$\$ \to \$ \to \$$	binäre Operatoren: $+, *, -, \wedge, \vee, \cap, \cup$
$\$ \to \$ \to \mathit{bool}$	Relationen: $=, \neq, \leq, >, <, >$

Definition 3.2 (Überladung). Sei ω ein Überladungsschema und τ ein Typausdruck, dann sagen wir τ *überlädt* ω *mit* τ', falls $\tau = \{\$ \mapsto \tau'\}\omega$ und $\tau' = f(\alpha_1, \ldots, \alpha_n)$ für ein $f \in F_n$ und $i \neq j \Rightarrow \alpha_i \neq \alpha_j$. In diesem Fall nennen wir τ' eine Überladung für ω.

Es ist genau diese Restriktion bzgl. der Form von τ', die es ermöglicht, einen Unifikationsalgorithmus für parametrische Überladungen zu entwickeln.

Definition 3.3 (Überladungsannahme). Eine Überladungsannahme O ist eine endliche Abbildung von Bezeichnern auf Paare der Form $\langle \omega, s \rangle$, wobei ω ein Überladungsschema ist und s eine Menge von Überladungen für ω, sodaß kein Typkonstruktor mehr als einmal in s vorkommt.

Aufgrund dieser Definition könnten wir nun die Menge der gültigen Überladungsinstanzen für überladene Operatoren $x : \langle \omega, s \rangle \in O$ wie folgt definieren:

$$\mathcal{I}_O(x) = \{t \mid t = \{\$ \mapsto f(\overline{t_n})\}\omega \wedge f(\overline{\alpha_n}) \in s\}$$

Die Überladungsannahme $\{+ : \langle \$ \to \$ \to \$, \{int, real\}\rangle\}$ würde dann die Menge $\{int \to int \to int, real \to real \to real\}$ spezifizieren, sodaß wir einfache Überladungen mit endlichen Mengen von Monotypen korrekt spezifizieren würden. Allerdings funktioniert die Methode nicht, wenn wir, wie im Fall des Gleichheitsoperators, eine unendliche Menge von Überladungen definieren möchten: mit der Überladungsannahme

$$\{= : \langle \$ \to \$ \to bool, \{int, real, list(\alpha)\}\rangle\}$$

hätten wir zwar ausgeschlossen, daß Funktionen als Argument von = auftreten können, aber Listen von Funktionen wären immer noch zulässig.

Um dieses Problem zu vermeiden, muß man, entsprechend der induktiven Definition der zulässigen Argumente von =, die Menge der möglichen Typen, die für α in $list(\alpha)$ ersetzt werden dürfen, auf die Menge der Typkonstruktoren beschränken, die in der Menge der Überladungen für = in O vorkommen.

Dazu verfeinern wir unseren Typvariablenbegriff: von einer unsortierten Menge von Typvariablen gehen wir über zu einer Familie abzählbar unendlicher Mengen von Variablen, die mit Mengen von Namen überladener Operatoren markiert sind:

$$\mathcal{V} = \bigcup_{X \subseteq dom(O)} V_X$$

Für $\alpha \in V_X$ schreiben wir auch kurz α_X oder $ops(\alpha) = X$. Dabei steht eine Variable α_X für die Menge der Typen, die für alle Operatoren aus X als Argumente an den überladenen Argumentpositionen verwendet werden können.

Mit dieser Erweiterung spezifiziert dann die Überladungsannahme

$$\{= : \langle \$ \to \$ \to bool, \{int, real, list(\alpha_{\{=\}})\}\rangle\}$$

die intendierte Menge der für = erlaubten Argumenttypen: nämlich all jene, die mit Hilfe der Typkonstruktoren *int*, *real* und *list* gebildet werden können.

Definition 3.4 ($x{\uparrow}\tau$, $X{\uparrow}\tau$, gültige Überladungen). Sei O eine Überladungsannahme. Der Typausdruck τ ist eine gültige Überladungsinstanz von x (geschrieben $x \uparrow^o \tau$) genau dann, wenn entweder $\tau \in \mathcal{V} \wedge x \in ops(\tau)$ oder $\tau = f(\tau_1, \ldots, \tau_n)$ und $O(x) = \langle \omega, s\rangle$ mit $f(\overline{\alpha_n}) \in s$ und $\forall i = 1..n : \forall y \in ops(\alpha_i) : y \uparrow^o \tau_i$. Für Mengen von Operatoren X definieren wir entsprechend: $X \uparrow^o \tau \equiv \forall x \in X.x \uparrow^o \tau$. Die Menge

der gültigen Überladungsinstanzen für $x \in dom(O)$ ist dann gegeben durch $\mathcal{I}_O(x) = \{\{\$ \mapsto t\}\omega | \{x\} \uparrow^o t\}$. Für $X \subseteq dom(O)$ definieren wir: $\mathcal{I}_O(X) = \bigcap_{x \in X} \mathcal{I}_O(x)$.

Beispiel: nehmen wir an, wir möchten die folgenden Bedingungen an die überladenen Operatoren $+$, $=$ und $\leq$ stellen: Gleichheit soll für ganze Zahlen, Fließkomma-Zahlen, Listen und Mengen definiert sein. $+$ soll mit Integer, Real-Addition und Mengenvereinigung überladen sein und $\leq$ soll für die arithmetische $\leq$-Relation, sowie für Listenpräfix und Teilmengen-Relation stehen. Diese Bedingungen werden von der folgenden Überladungsannahme spezifiziert:

$$
\begin{aligned}
+ &: \$ \to \$ \to \$, \{int, real, set(\alpha_{\{=\}})\} \\
= &: \$ \to \$ \to bool, \{int, real, list(\alpha_{\{=\}}), set(\alpha_{\{=\}})\} \\
\leq &: \$ \to \$ \to bool, \{int, real, list(\alpha_{\{=\}}), set(\alpha_{\{=\}})\}
\end{aligned}
$$

Man überzeugt sich leicht, daß die obige Definition von $\uparrow^0$ unserer Intuition über überladene Operatoren entspricht:

Proposition 3.5. *Sei O eine Überladungsannahme und $X, Y \subseteq dom(O)$. Dann gilt:*

$$
\begin{aligned}
X \uparrow^0 \tau \wedge Y \uparrow^0 \tau &\Rightarrow (X \cup Y) \uparrow^0 \tau \\
X \uparrow^0 \tau \wedge Y \subseteq X &\Rightarrow Y \uparrow^0 \tau
\end{aligned}
$$

Beweis. Folgt unmittelbar aus der Definition von $\uparrow^0$. □

In Tabelle 3.3 sind die überladenen Operatoren für die Sprache SAMPλE zusammengefaßt. $\times_n$ bezeichnet dabei den Typkonstruktor für das n-fache kartesische Produkt (n-Tupel). Die meisten Funktionen sind selbsterklärend, die Funktion *show* dient der Umwandlung beliebiger Daten in textuelle Repräsentationen und ist auf Daten, die funktionale Werte beinhalten, nicht definiert. Die Funktion $<$ wurde so definiert, daß die folgende Semantik implementiert werden kann: Für Listen gilt $\langle x_1, \ldots, x_n \rangle < \langle y_1, \ldots, y_m \rangle$, falls $(n < m \wedge \forall i \leq n : x_i = y_i)$ oder $(\exists j \leq min(m, n) : x_j < y_j \wedge \forall i < j : x_i = x_j)$; damit ergibt sich für den Typ $list(char)$ die lexikographische Ordnung. Die Bedeutung von $<$ für Mengen und endliche Abbildungen wird zurückgeführt auf die Überladungen der Funktionen $=$ und $<$ für Listen und Paare:

$$
\begin{aligned}
s_1 < s_2 &\iff sort(elems\, s_1) < sort(elems\, s_2) \\
m_1 < m_2 &\iff sort(pairs\, m_1) < sort(pairs\, m_2)
\end{aligned}
$$

$$
\begin{array}{ll}
= : & \$ \to \$ \to \mathit{bool}, \\
 & \{\ \mathit{int}, \mathit{real}, \mathit{bool}, \mathit{char}, \mathit{ref}(\alpha), \mathit{enum}(\overline{\alpha_n}), \\
 & \quad \mathit{list}(\alpha_{\{=\}}), \mathit{set}(\alpha_{\{=\}}), \mathit{map}(\alpha_{\{=\}}, \beta_{\{=\}}), \\
 & \quad \times_2(\alpha_{\{=\}}, \beta_{\{=\}}), \times_3(\alpha_{\{=\}}, \beta_{\{=\}}, \gamma_{\{=\}}), \ldots\ \} \\
< : & \$ \to \$ \to \mathit{bool}, \\
 & \{\ \mathit{int}, \mathit{real}, \mathit{bool}, \mathit{char}, \mathit{enum}(\overline{\alpha_n}), \\
 & \quad \mathit{list}(\alpha_{\{<,=\}}), \mathit{set}(\alpha_{\{<,=\}}), \mathit{map}(\alpha_{\{<,=\}}, \beta_{\{<,=\}}), \\
 & \quad \times_2(\alpha_{\{<,=\}}, \beta_{\{<\}}), \times_3(\alpha_{\{<,=\}}, \beta_{\{<,=\}}, \gamma_{\{<\}}), \ldots\ \} \\
+ : & \$ \to \$ \to \$, \{\mathit{int}, \mathit{real}\} \\
- : & \$ \to \$ \to \$, \{\mathit{int}, \mathit{real}\} \\
* : & \$ \to \$ \to \$, \{\mathit{int}, \mathit{real}\} \\
\mathrm{div} : & \$ \to \$ \to \$, \{\mathit{int}\} \\
\mathrm{mod} : & \$ \to \$ \to \$, \{\mathit{int}\} \\
/ : & \$ \to \$ \to \$, \{\mathit{real}\} \\
\mathrm{minus} : & \$ \to \$, \{\mathit{int}, \mathit{real}\} \\
\mathrm{abs} : & \$ \to \$, \{\mathit{int}, \mathit{real}\} \\
\mathrm{ord} : & \$ \to \mathit{int}, \{\mathit{int}, \mathit{enum}(\overline{\alpha_n})\} \\
\mathrm{succ} : & \$ \to \$, \{\mathit{int}, \mathit{enum}(\overline{\alpha_n})\} \\
\mathrm{pred} : & \$ \to \$, \{\mathit{int}, \mathit{enum}(\overline{\alpha_n})\} \\
\mathrm{show} : & \$ \to \mathit{list}(\mathit{char}) \to \mathit{list}(\mathit{char}), \\
 & \{\ \mathit{int}, \mathit{real}, \mathit{bool}, \mathit{char}, \mathit{unit}, \mathit{ref}(\alpha_{\{\mathrm{show}\}}), \mathit{enum}(\overline{\alpha_n}), \\
 & \quad \mathit{list}(\alpha_{\{\mathrm{show}\}}), \mathit{set}(\alpha_{\{\mathrm{show}\}}), \mathit{map}(\alpha_{\{\mathrm{show}\}}, \beta_{\{\mathrm{show}\}}), \\
 & \quad \times_2(\alpha_{\{\mathrm{show}\}}, \beta_{\{\mathrm{show}\}}), \times_3(\alpha_{\{\mathrm{show}\}}, \beta_{\{\mathrm{show}\}}, \gamma_{\{\mathrm{show}\}}), \ldots\ \}
\end{array}
$$

Abbildung 3.3: Vordefinierte überladene Operatoren für SAMPλE

wobei die Funktion *elems* eine Liste der Elemente einer Menge, die Funktion *pairs* eine Liste der Zuordnungen einer endlichen Abbildung liefert und die Funktion *sort* eine Liste mit Hilfe der überladenen Operatoren = und < sortiert.

Nach diesen positiven Beispielen betrachten wir kurz, welche Arten von Überladungen nicht parametrisch sind: das sind all diejenigen, deren mögliche Instanztypen keine induktive Definition wie für den Gleichheitsoperator erlauben. So können wir zwar die arithmetischen Operatoren so überladen, daß für alle Parameter der gleiche Typ verwendet kann $\{int \to int \to int, real \to real \to real\}$, aber wir können die Parametertypen nicht unabhängig voneinander instanzieren, sodaß z.B. $int \to real \to real$ und $int \to real \to real$ zulässig wären. Wir werden später in Kapitel 6 sehen, daß dieses spezielle Beispiel durch implizite Konversionen behandelt werden kann. Eine weniger triviales Beispiel ist die Matrix-Multiplikation: zwar können wir mit der Überladungsannahme

$$+ : \$ \to \$ \to \$, \{int, real, matrix(\alpha_{\{+\}})\}$$
$$* : \$ \to \$ \to \$, \{int, real, matrix(\alpha_{\{+,*\}})\}$$

spezifizieren, daß die Matrixmultiplikation auf allen Matrizen erlaubt ist, deren Elemente sowohl Addition als auch Multiplikation erlauben, wir können jedoch nicht sicherstellen, daß die Dimensionen der betrachteten Matrizen korrekt zueinander passen. Dazu müßte man die Dimensionen im Typkonstruktor $matrix(\alpha, n, m)$ sichtbar machen und die zulässigen Überladungsinstanzen über einen Typausdruck der Form $\forall \alpha_{\{*,+\}}, n, m.matrix(\alpha, n, r) \to matrix(\alpha, r, m) \to matrix(\alpha, n, m)$ beschreiben können, was aber mit parametrischen Überladungen aufgrund der Einschränkung auf identische Typen an überladenen Argumentpositionen nicht möglich ist.[2]

3.2 Gültige Typisierungen

Zur Definition eines Typdeduktionssystems benötigen wir 2 Dinge: eine Repräsentation der Menge der möglichen Typinstanzen eines überladenen Operators in Form eines Typschemas in der Typumgebung Γ, unter der wir den Typ eines Ausdrucks inferieren, und eine Definition der generischen Instanz von Typausdrücken.

[2] Allerdings kann diese Überladungsrestriktion mit Hilfe der in Kapitel 4 beschriebenen *eingeschränkten Typen* dargestellt werden.

$$\text{[VAR]} \quad \frac{\tau \prec^o \Gamma(x)}{\Gamma \vdash^o x : \tau}$$

$$\text{[ABS]} \quad \frac{\Gamma + [x : \tau] \vdash^o M : \tau' \qquad x \notin dom(O)}{\Gamma \vdash^o \lambda x.M : \tau \to \tau'}$$

$$\text{[APP]} \quad \frac{\Gamma \vdash^o M_1 : \tau \to \tau' \,,\, \Gamma \vdash^o M_2 : \tau}{\Gamma \vdash^o M_1 M_2 : \tau'}$$

$$\text{[LET]} \quad \frac{\Gamma \vdash^o M_1 : \tau \,,\, \Gamma + [x : \mathrm{gen}(\Gamma, \tau)] \vdash^o M_2 : \tau' \qquad x \notin dom(O)}{\Gamma \vdash^o \textbf{let } x = M_1 \textbf{ in } M_2 : \tau'}$$

Abbildung 3.4: Typdeduktionssystem für parametrische Überladungen

Den polymorphen Typ eines überladenen Operators $x \in dom(O)$ repräsentieren wir durch das Typschema, das man erhält, wenn man das Symbol \$ durch eine nicht in ω vorkommende neue Variable $\alpha_{\{x\}}$ ersetzt, und alle freien Variablen des resultierenden Typausdrucks abstrahiert.

Definition 3.6. Sei Γ eine Typannahme und O eine gültige Überladungsannahme. Γ *stimmt mit* O *überein*, falls $x : \langle \omega, s \rangle \in O$ und $x : \sigma \in \Gamma$, dann gilt $\sigma = \forall \alpha_{\{x\}}.\forall \overline{\beta_n}.\{\$ \mapsto \alpha_{\{x\}}\}\omega$, wobei $\overline{\beta_n} = \mathcal{V}(\omega)$ und $\alpha_{\{x\}} \notin \mathcal{V}(\omega)$.

Generische Instanz definieren wir so, daß für α_X nur Typausdrücke τ substituiert werden dürfen, die $X \uparrow^o \tau$ erfüllen:

Definition 3.7. Sei O eine Überladungsannahme. Ein Typschema σ' heißt *generische (Überladungs-)Instanz* eines Typschemas σ, geschrieben $\sigma' \prec^o \sigma$, falls $\sigma' = \forall \beta_1, \ldots, \beta_m.\tau'$ und $\sigma = \forall \alpha_1, \ldots, \alpha_n.\tau$, $\tau' = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}\tau$, keines der β_i frei in τ vorkommt und außerdem $\forall i \in 1..n : ops(\alpha_i) \uparrow^o \tau_i$.

Das Deduktionssystem in Abbildung 3.4 definiert die gültigen Typisierungen in der Gegenwart eine Überladungsannahme O. Der einzige Unterschied zum Damas-Milner-System liegt in der Verwendung der generischen Instanzrelation $\prec^o$.

Beispiel: Mit dem obigen Deduktionssystem kann man unter der Typannahme

$$\Gamma = \{+ : \forall\alpha_{\{+\}}.\alpha_{\{+\}} \to \alpha_{\{+\}} \to \alpha_{\{+\}}, \quad * : \forall\alpha_{\{*\}}.\alpha_{\{*\}} \to \alpha_{\{*\}} \to \alpha_{\{*\}}\}$$

und der Überladungsannahme

$$O = \{+ : \langle \$ \to \$ \to \$, \{int, real\}\rangle, \quad * : \langle \$ \to \$ \to \$, \{int, real\}\rangle\}$$

ableiten, daß

$$\Gamma \vdash^o \textbf{let } f = \lambda x.x + x * x \textbf{ in } (f\ 5) : int$$

eine gültige Typisierung ist. Man beachte, daß dem deklarierenden Vorkommen von f das Typschema

$$\forall\alpha_{\{+,*\}}.\alpha_{\{+,*\}} \to \alpha_{\{+,*\}}$$

zugeordnet wird, das wie folgt interpretiert werden kann: f ist eine überladene Funktion, die auf Werte angewendet werden kann, deren Typ sowohl für $+$ als auch $*$ als Argumenttyp erlaubt ist. Der Typ von f in der Applikation $(f\ 5)$ wird dann mit $int \to int$ instanziert, da int eine gültige Instanzierung für $\alpha_{\{+,*\}}$ ist.

3.3 Semantik von Typen und Ausdrücken

Vordefinierte überladene Operatoren sind ganz normale Funktionen in dem betrachteten semantischen Bereich **DV**. Beispiel: die Semantik einer überladenen Additionsoperation für ganze Zahlen und Fließkommazahlen:

$$\begin{aligned}[\![+]\!] = \lambda a.\lambda b.\ & a \subseteq \boldsymbol{D}_{int} \wedge b \subseteq \boldsymbol{D}_{int} \to intadd\ a\ b,\\ & a \subseteq \boldsymbol{D}_{real} \wedge b \subseteq \boldsymbol{D}_{real} \to realadd\ a\ b,\\ & wrong\end{aligned}$$

Hier gehen wir davon aus, daß entsprechende unüberladene Additionsoperationen für die elementaren Typen existieren. Etwas komplexer ist die Definition eines überladenen Gleichheitsoperators:

$$\begin{aligned}[\![=]\!] = Y\lambda f.\lambda a.\lambda b.\ & a \subseteq \boldsymbol{D}_{int} \wedge b \subseteq \boldsymbol{D}_{int} \to inteq\ a\ b,\\ & a \subseteq \boldsymbol{D}_{real} \wedge b \subseteq \boldsymbol{D}_{real} \to realeq\ a\ b,\\ & a \subseteq \boldsymbol{D}_{\times} \wedge b \subseteq \boldsymbol{D}_{\times} \to f\ (fst\ a)\ (fst\ b) \wedge f\ (snd\ a)\ (snd\ b),\\ & wrong\end{aligned}$$

Hier benötigt man eine rekursive Funktionsdefinition, bzw. die Anwendung des Fixpunktoperators Υ, um die Definiertheit auf beliebig komplexen Typausdrücken, die mit den Typkonstruktoren *int*, *real* und $\times$ gebildet werden können, zu garantieren.

In beiden Fällen ist intuitiv klar, daß die Menge der zulässigen Argumenttypen durch eine Überladungsannahme beschrieben werden kann. Außerdem ist es nicht notwendig die Semantik von Ausdrücken gegenüber dem rein parametrisch polymorphen Fall (siehe Seite 32) zu ändern: die Feststellung, ob ein Argument zulässig ist, wird implizit durch die Semantik der überladenen Operatoren festgelegt.

Zum Beweis der Wohldefiniertheit unseres Typdeduktionssystems benötigen wir jedoch eine Anpassung der Semantik von Typausdrücken, welche die Markierung von Typvariablen durch Operatormengen und die Abhängigkeit von der Überladungsannahme O berücksichtigt.

Wir beginnen mit einem Beispiel: angenommen der Additionsoperator + sei mit der Ganzzahl- und Fließkommaaddition überladen, dann soll $f = \lambda x.x + x$ eine Funktion sein, die sowohl auf ganze Zahlen, als auch auf Fließkommazahlen angewendet werden kann, wobei das Ergebnis fx vom gleichen Typ wie das Argument ist. Mit anderen Worten: f liegt in der Schnittmenge der Ideale $\mathcal{T}[\![int \to int]\!]$ und $\mathcal{T}[\![real \to real]\!]$. Daher definieren die Semantik von Typausdrücken so, daß $\mathcal{T}[\![\forall\alpha_{\{+\}}.\alpha_{\{+\}} \to \alpha_{\{+\}}]\!] \subseteq \mathcal{T}[\![int \to int]\!] \cap \mathcal{T}[\![real \to real]\!]$.

Definition 3.8. Sei O eine Überladungsannahme und φ eine Abbildung, die den freien Variablen in σ Ideale aus $\mathbf{I}(\mathbf{DV})$ zuordnet, sodaß

$$\alpha_X \in dom(\varphi) \Rightarrow \exists t.X \uparrow^o t \wedge \varphi(\alpha_X) = \mathcal{T}[\![t]\!]\emptyset \tag{3.1}$$

dann definieren wir $\mathcal{T}_o[\![]\!]$ wie folgt:

$$\begin{aligned}
\mathcal{T}_o[\![\alpha_X]\!]\varphi &= \varphi(\alpha_X) \\
\mathcal{T}_o[\![f(\tau_1,\ldots,\tau_n)]\!]\varphi &= \mathbf{i}(f)(\mathcal{T}_o[\![\tau_1]\!]\varphi,\ldots,\mathcal{T}_o[\![\tau_n]\!]\varphi) \\
\mathcal{T}_o[\![\forall\alpha_X.\sigma]\!]\varphi &= \textstyle\bigcap_{t\in T_F(\emptyset)\wedge X\uparrow^o t} \mathcal{T}_o[\![\sigma]\!]\varphi\{\mathcal{T}_o[\![t]\!]\varphi/\alpha_X\}\,.
\end{aligned}$$

Damit $\mathcal{T}_o[\![]\!]$ wohldefiniert ist, müssen wir fordern, daß für jede Menge von Operatornamen $X \subseteq dom(O)$, die Menge aller Typausdrücke aus $T_F(\mathcal{V}_\emptyset)$ nicht leer ist:

$$\{\tau \mid X \uparrow^o \tau \wedge \mathcal{V}(\tau) \subseteq \mathcal{V}_\emptyset\} \neq \emptyset \tag{3.2}$$

denn sonst wäre $\mathcal{T}_o[\![\forall\alpha_X.\sigma]\!]$ u.U. undefiniert.

Lemma 3.9. *Sei O eine Überladungsannahme, $\sigma = \forall\overline{\alpha_n}.\tau$ ein Typschema, sodaß für alle $ops(\alpha_i)$ die Bedingung 3.2 erfüllt ist und φ eine Typzuweisung, die Bedingung 3.1 erfüllt, dann ist $\mathcal{T}_o[\![\sigma]\!]$ ein Ideal.*

Beweis. Durch Induktion über σ. Für $\sigma = \alpha_X$ kommt die Voraussetzung an φ zur Anwendung. Für den Fall $\sigma = f(\tau_1, \ldots, \tau_n)$ folgt die Behauptung aus der Induktionshypothese und der Tatsache daß $\mathbf{i}(f)$ ein Idealkonstruktor ist. Für $\sigma = \forall\alpha_X.\sigma'$ ist $\bigcap_{t\in T_F(\emptyset)\wedge X\uparrow^o t} \mathcal{T}_o[\![\sigma']\!]\varphi\{\mathcal{T}_o[\![t]\!]\varphi/\alpha_X\}$ ein Ideal, da die Schnittmenge von Idealen wiederum ein Ideal ist und die Induktionshypothese angewendet werden kann, da $\varphi\{\mathcal{T}_o[\![t]\!]\varphi/\alpha_X\}$ Bedingung 3.1 erfüllt. □

Bedingung 3.2 kann man auf mehrere Arten erfüllen.

Zum Einen kann die Überladungsannahme schon so beschaffen sein, daß diese Bedingung immer erfüllt ist. Im einfachsten Fall gibt es einen Basistyp, für den alle überladenen Operatoren definiert sind:

$$\exists f \in F_0.\forall x \mapsto (\omega, s) \in O.f() \in s\ . \tag{3.3}$$

Wie man leicht sieht, gilt dies für die in Abbildung 3.3 vordefinierten überladenen Operatoren von SAMPλE nicht, da z.B. *div* und / keinen gemeinsamen Instanztyp besitzen. Dennoch läßt sich für die Frage nach der Existenz einer Lösung mit einem $O(n)$-Algorithmus entscheiden, wobei n die Anzahl der Knoten plus die Anzahl der Kanten der DAG-Repräsentation des Terms τ ist. Dazu traversiert man den Termgraphen und berechnet für jede Typvariable α_X die Schnittmenge $\bigcap_{x\in X}\{f \in F_0 \mid x\uparrow^o f\}$. Ist keine der Schnittmengen leer, dann gibt es eine gültige Überladungsinstanz, da für SAMPλE die folgende Bedingung erfüllt ist:

$$\mathcal{I}_O(X) \neq \emptyset \iff \exists f \in F_0 : X\uparrow^o f \tag{3.4}$$

In Worten: ein Typausdruck τ hat genau denn eine überladungsfreie Instanz, wenn er eine überladungsfreie Basistypinstanz besitzt.

Sollte Bedingung 3.3 nicht erfüllt sein, kann man auch einen neuen Null-stelligen Typkonstruktor definieren, dessen zugeordnetes Ideal in **DV** einen einzigen Wert

enthält. In funktionalen Programmiersprachen mit Seiteneffekten ist ein passender Typkonstruktor *unit* üblicherweise bereits vorhanden. Der einzige Wert wird dabei zumeist mit () notiert. Definiert man nun für jeden Operator die Semantik so, daß () ein zulässiges Argument ist, dann ist Bedingung 3.3 klarerweise erfüllt und jedes Programm hat eine eindeutig definierte Semantik.

Bedingung 3.3 ist zwar hinreichend, aber nicht notwendig, um 3.2 zu erfüllen, wie man an folgendem Beispiel sieht:

$$x : \$ \to \$, \{int, list(\alpha_{\{x\}})\}$$
$$y : \$ \to \$, \{real, list(\alpha_{\{x\}})\}$$

Zwar gibt es keinen Basistyp, der für jeden Operator als Argument erlaubt ist, aber der Typausdruck $list(int)$ ist offensichtlich in $\mathcal{I}_O(x) \cap \mathcal{I}_O(y)$. Allgemein gilt:

Satz 3.10. *Sei O eine Überladungsannahme und $X \subseteq dom(O)$. Es ist entscheidbar, ob $\mathcal{I}_O(X)$ mindestens ein Element enthält.*

Beweis. Wir konstruieren eine kontextfreie Grammatik $G_O = (T, N, P, S)$ mit Terminalen T, Nichtterminalen N, Produktionen P und Startsymbol S, sodaß $\mathcal{I}_O(X) \neq \emptyset$ genau dann, wenn die von G_O erzeugte Sprache $L(G_O)$ nicht leer ist. Da das Leerheitsproblem für kontextfreie Grammatiken entscheidbar ist, folgt daraus die Behauptung. Dazu wählen wir[3]

$$T = F \cup \{\, \emptyset\,,\ \texttt{','},\ \texttt{'('},\ \texttt{')'} \,\}$$
$$N = \mathbf{2}^{dom(O)} - \emptyset$$

sowie

$$P = \{\, Y \to \texttt{f (} Y_1 \texttt{ , } \ldots \texttt{ , } Y_n \texttt{)} \mid \exists f \in F.Y \subseteq m(f) \wedge Y_i = d_f(i, Y) \,\}$$
$$S = X$$

Somit werden Variablen $\alpha \in \mathcal{V}_\emptyset$ durch das Terminal $\emptyset$ und Variablen $\alpha \in \mathcal{V} - \mathcal{V}_\emptyset$ durch Nichtterminale repräsentiert. Jedem Typausdruck τ mit $\mathcal{V}(\tau) \subseteq \mathcal{V}_\emptyset$ kann man eineindeutig ein Wort $w \in L_G$ zuordnen, sodaß $X \uparrow^o \tau \iff X \stackrel{*}{\Longrightarrow} w$. Der Beweis in

[3] Terminale setzen wir zur Unterscheidung von Metasymbolen im Schreibmaschinenzeichensatz: `Schreibmaschine`.

$\Rightarrow$-Richtung erfolgt durch Induktion über τ, die $\Leftarrow$-Richtung durch Induktion über die Länge der Ableitung $\overset{*}{\Longrightarrow}$. □

Aufgrund der Konstruktion der Grammatik ist klar, daß die Verifikation der Frage $\mathcal{I}_O(X) = \emptyset$? exponentiellen Aufwand erfordern kann: zwar ist das Leerheitsproblem mit polynomiellen Zeitaufwand in der Anzahl der Produktionen lösbar, aber die o.a. Konstruktion erzeugt 2^n Produktionen, wobei $n = |dom(O)|$ die Anzahl der überladenen Operatoren ist. Trotz intensiver Bemühungen konnte der Autor jedoch keinen besseren Algorithmus finden. Das dies nicht an fehlendem Erfindergeist lag, wurde später von Volpano und Seidl bewiesen: zunächst zeigte Volpano [Vol94], daß die Lösungssuche NP-schwer ist, später bewies Seidl in [Sei94] durch Reduktion auf die Schnittmengenbildung von deterministischen Top-down Baumautomaten, daß dieses Problem deterministisch Exponentialzeit vollständig ist (DEXPTIME-complete). Dieses Resultat wurde noch später ergänzt von Volpano [Vol96]: fordert man, daß für $x \mapsto (\omega, s) \in O$ die Menge der Typausdrücke in s auf Typkonstruktoren aus $F_{0+1} = F_0 \cup F_1$ beschränkt wird ($\tau \in s \Rightarrow \tau = f(\overline{\alpha_n}) \wedge n \leq 1$), dann bleibt die Frage $\mathcal{I}_O(X) = \emptyset$? Polynomialplatz vollständig (PSPACE-complete). Selbst bei einer Beschränkung auf nicht rekursive Überladungsannahmen und Typkonstruktoren aus F_{0+1} bleibt das Problem NP-schwer.

Von Geoffrey Smith [VS91, Smi94] wurde eine Einschränkung der parametrischen Überladungen vorgeschlagen, die einen polynomial zeitbeschränkten Test der Frage $\mathcal{I}_O(X)$ ermöglicht. Smith verlangt zum einen, daß die Semantik rekursiv definierter überladener Operatoren keine gegenseitigen Abhängigkeiten aufweist ($\leq$ darf also z.B. nicht auf $=$ zurückgeführt werden) und zum zweiten fordert er, daß die Menge der zulässigen Argumenttypen eines überladenen Operators allein durch Angabe der zulässigen Typkonstruktoren spezifiziert werden kann (z.B. $=$ ist erlaubt auf allen Typen, die mit Hilfe der Typkonstruktoren $int, real, list, set, \times$ gebildet werden können). Smith nennt diese beiden Forderungen *Typkonstruktoreigenschaft*.

Übersetzt in das System der parametrischen Überladungen bedeutet dies, daß für jeden Operator x die folgende Bedingung erfüllt ist:

$$O(x) = \langle \omega, s \rangle \Rightarrow f(\overline{\alpha_n}) \in s \Rightarrow \forall i = 1..n : ops(\alpha_i) = \{x\} \qquad \text{(tke)}$$

$$\text{[VAR]} \quad \frac{\tau \prec^o \Gamma(x)}{\Gamma \vdash^o x : \tau \xrightarrow{\text{gtv}} \mathcal{V}(\tau) - \mathcal{FV}(\Gamma)}$$

$$\text{[ABS]} \quad \frac{\Gamma + [x:\tau] \vdash^o M : \tau' \xrightarrow{\text{gtv}} V}{\Gamma \vdash^o \lambda x.M : \tau \to \tau' \xrightarrow{\text{gtv}} V \cup (\mathcal{V}(\tau) - \mathcal{FV}(\Gamma))}$$

$$\text{[APP]} \quad \frac{\Gamma \vdash^o M_1 : \tau \to \tau' \xrightarrow{\text{gtv}} V_1 \ , \ \Gamma \vdash^o M_2 : \tau \xrightarrow{\text{gtv}} V_2}{\Gamma \vdash^o M_1 M_2 : \tau' \xrightarrow{\text{gtv}} V_1 \cup V_2}$$

$$\text{[LET]} \quad \frac{\Gamma \vdash^o M_1 : \tau \xrightarrow{\text{gtv}} V_1 \ , \ \Gamma + [x : \text{gen}(\Gamma, \tau)] \vdash^o M_2 : \tau' \xrightarrow{\text{gtv}} V_2}{\Gamma \vdash^o \textbf{let } x = M_1 \textbf{ in } M_2 : \tau' \xrightarrow{\text{gtv}} V_1 \cup V_2}$$

Abbildung 3.5: Deduktion generischer Typvariablen

Proposition 3.11. *Falls Bedingung (tke) für alle $x \in dom(O)$ erfüllt ist, dann gilt $\mathcal{I}_O(X) = \emptyset \Rightarrow \exists f \in F_0 : f \in \mathcal{I}_O(X)$.*

Beweis. Sei $\mathcal{I}_O(X) \neq \emptyset$. Dann existiert ein $f(\overline{\tau_n}) \in \mathcal{I}_O(X)$ mit minimaler Termhöhe $h = |f(\overline{\tau_n})|$.[4] Für $h = 1$ ist die Behauptung erfüllt. Falls $h > 1$ dann gilt aber aufgrund der Definition von $\uparrow^o$ und (tke): $X \uparrow^o f(\overline{\tau_n}) \Leftrightarrow \forall i = 1..n : X \uparrow^o \tau_i$. Somit gilt $\tau_i \in \mathcal{I}_O(X)$. Da aber $|\tau_i| < h$, steht dies im Widerspruch zur Minimalität von h. □

Daraus folgt sofort, daß $\mathcal{I}_O(X) = \emptyset$? mit polynomiellen Zeitaufwand entschieden werden kann: berechne die Menge $M_X = \{f \mid X \subseteq m(f) \wedge f \in F_0\}$. Falls M_X leer ist, gilt $\mathcal{I}_O(X) = \emptyset$.

Man beachte jedoch, daß z.B. die vordefinierten Operatoren von SAMPλE die Typkonstruktoreigenschaft nicht erfüllen.[5] Daher beurteilt der Autor die Typkonstruktoreigenschaft als zu restriktiv für den praktischen Einsatz.

[4] Die Höhe $|\tau|$ ist dabei rekursiv definiert durch: $|\alpha| = 1$, $|f| = 1$ für $\rho(f) = 0$, $|f(\overline{\tau_n})| = 1 + \max\{|\tau_i| \mid i \in 1..n\}$ für $\rho(f) > 0$.

[5] Der Operator = ist auf allen Referenzzellen $ref(\alpha)$ definiert, unabhängig vom Typ der referenzierten Elemente.

Für den Beweis der Aussage, daß Überladungstypisierung Laufzeittypfehler verhindert, benötigen wir eine formale Definition der generischen Typvariablen einer Typisierung. Dies erledigen wir mit Hilfe des Deduktionssystem, das in Abbildung 3.5 angegeben ist.

Satz 3.12 (Überladungstypisierung verhindert Laufzeitfehler). *Sei O eine Überladungsannahme, Γ stimme mit O überein und $\Gamma \vdash^o M : \tau$ sei eine gültige Typisierung, sodaß $\Gamma \vdash^o M : \tau \xrightarrow{\text{gtv}} V$ und $\mathcal{I}_O(X) \neq \emptyset$ für alle $\alpha_X \in V$. Falls $x : \sigma \in \Gamma \Rightarrow \eta(x) \in \mathcal{T}_o[\![\sigma]\!]\varphi$ dann gilt $\mathcal{E}[\![M]\!]\eta \in \mathcal{T}_o[\![\text{gen}(\Gamma, \tau)]\!]\varphi$.*

Beweis. Die Bedingung $\mathcal{I}_O(X) \neq \emptyset$ für alle $\alpha_X \in V$ mit $\Gamma \vdash^o M : \tau \xrightarrow{\text{gtv}} V$ stellt sicher, daß für jeden Teilausdruck von M', mit der Typisierung $\Gamma' \vdash^o : \tau'$, die Sorten aller in $\text{gen}(\Gamma', \tau')$ vorkommenden Typvariablen nicht überladene Instanztypen erlauben. Damit ist $\mathcal{T}[\![\text{gen}(\Gamma', \tau')]\!]$ immer wohldefiniert, die Propositionen 2.31, 2.32, 2.33 und 2.34 sind anwendbar und der Beweis von Satz 2.35 läßt sich direkt übertragen. □

Kann der Sprachdesigner nicht garantieren, daß für jede Menge von überladenen Operatoren mindestens ein Typ existiert, der für alle Operatoren als Argumenttyp zulässig ist, dann wird er wohl oder übel nicht umhin kommen, den Leerheitstest $\mathcal{I}_O(X) =^? \emptyset$ zu implementieren und riskiert intolerable Laufzeiten für den Typinferenzalgorithmus. Glücklicherweise kann man jedoch durch eine weitere Bedingung an die Semantik der vordefinierten überladenen Operatoren garantieren, daß dieser Test niemals durchgeführt werden muß.

Betrachten wir dazu den Ausdruck $M = \lambda x.x + x * x$ unter der Überladungsannahme

$$O = [\, + : \$ \to \$ \to \$, \{int\} \ , \ * : \$ \to \$ \to \$, \{real\} \,]$$

Unter der zu O kompatiblen Typannahme

$$\Gamma = [\, + : \forall\alpha_{\{+\}}.\alpha_{\{+\}} \to \alpha_{\{+\}} \to \alpha_{\{+\}} \ , \ * : \forall\alpha_{\{*\}}.\alpha_{\{*\}} \to \alpha_{\{*\}} \to \alpha_{\{*\}} \,]$$

ist $\Gamma \vdash M : \alpha_{\{+,*\}} \to \alpha_{\{+,*\}} \xrightarrow{\text{gtv}} \{\alpha_{\{+,*\}}\}$ eine gültige Typisierung, wobei offensichtlich $\mathcal{I}_O(\{+, *\}) = \emptyset$ gilt, d.h. es gibt keinen gemeinsam zulässigen Argumenttyp.

Falls wir für $+$ und $*$ die folgende Semantik annehmen:

$$[\![+]\!] = \lambda a.\lambda b. \begin{cases} a +_{int} b & \text{falls } a \in \boldsymbol{D}_{int} \wedge b \in \boldsymbol{D}_{int}, \\ \bot & \text{falls } a = \bot \vee b = \bot, \\ \mathit{wrong} & \text{sonst.} \end{cases}$$

$$[\![*]\!] = \lambda a.\lambda b. \begin{cases} a *_{real} b & \text{falls } a \in \boldsymbol{D}_{real} \wedge b \in \boldsymbol{D}_{real}, \\ \bot & \text{falls } a = \bot \vee b = \bot, \\ \mathit{wrong} & \text{sonst.} \end{cases}$$

dann sieht man, daß es dennoch einen Wert $v \in \mathbf{DV}$ gibt, der verschieden von *wrong* ist und als Argument beider Operatoren zulässig ist: $\bot$! Durch Anwenden der Semantikfunktion $\mathcal{E}[\![]\!]$ und Umformung erhält man:

$$[\![\lambda x.x + x * x]\!] = \lambda v. \begin{cases} \bot & \text{falls } v = \bot, \\ \mathit{wrong} & \text{sonst.} \end{cases}$$

Definition 3.13. Sei O eine Überladungsannahme und φ eine Abbildung, die den freien Variablen in σ Ideale aus $\mathbf{I}(\mathbf{DV})$ zuordnet, sodaß

$$\alpha_X \in dom(\varphi) \Rightarrow \exists t.X \uparrow^o t \wedge \varphi(\alpha_X) = \mathcal{T}[\![t]\!]\emptyset \tag{3.5}$$

dann definieren wir $\mathcal{T}_o^\bot[\![]\!]$ wie folgt:

$$\begin{aligned} \mathcal{T}_o^\bot[\![\alpha_X]\!]\varphi &= \varphi(\alpha_X) \\ \mathcal{T}_o^\bot[\![f(\tau_1,\ldots,\tau_n)]\!]\varphi &= \mathbf{i}(f)(\mathcal{T}_o^\bot[\![\tau_1]\!]\varphi,\ldots,\mathcal{T}_o^\bot[\![\tau_n]\!]\varphi) \\ \mathcal{T}_o^\bot[\![\forall\alpha_X.\sigma]\!]\varphi &= \begin{cases} \mathcal{T}_o^\bot[\![\sigma]\!]\varphi\{\{\bot\}/\alpha_X\} & \text{falls } \mathcal{I}_O(X) = \emptyset, \\ \bigcap_{t\in T_F(\emptyset)\wedge X\uparrow^o t} \mathcal{T}_o^\bot[\![\sigma]\!]\varphi\{\mathcal{T}_o^\bot[\![t]\!]\varphi/\alpha_X\} & \text{falls } \mathcal{I}_O(X) \neq \emptyset. \end{cases} \end{aligned}$$

Proposition 3.14. *$\mathcal{T}_o^\bot[\![\sigma]\!]\varphi$ ist wohldefiniert, d.h $\mathcal{T}_o^\bot[\![\sigma]\!]\varphi \in \mathbf{I}(\mathbf{DV})$ falls $\alpha \in \mathcal{V}(\sigma) \Rightarrow \varphi(\alpha) \in \mathbf{I}(\mathbf{DV})$.*

Satz 3.15 (Typisierung verhindert Laufzeitfehler). *Sei O eine Überladungsannahme, Γ stimme mit O überein und $\Gamma \vdash^o M : \tau$ sei eine gültige Typisierung. Falls $x : \sigma \in \Gamma \Rightarrow \eta(x) \in \mathcal{T}_o^\bot[\![\sigma]\!]\varphi$ erfüllt ist, gilt $\mathcal{E}[\![M]\!]\eta \in \mathcal{T}_o^\bot[\![\mathrm{gen}(\Gamma,\tau)]\!]\varphi$.*

Beweis. Wie man leicht verifiziert, gelten die Propositionen 2.31, 2.32, 2.33 und 2.34 auch für $\mathcal{T}_o^\bot$. Daher kann der Beweis von Satz 2.35 für den nicht-überladenen Fall unter Anwendung von Proposition 3.14 direkt übertragen werden. □

3.4 Typinferenz für vordefinierte überladene Operatoren

Für die Entwicklung eines Typinferenzalgorithmus für Ausdrücke mit parametrisch überladenen Operatoren benötigen wir einen Unifikationsalgorithmus für Typausdrücke mit überladungssortierten Typvariablen, den wir im Folgenden entwickeln werden. Die Existenz eindeutiger allgemeinster Typen folgt dann sofort durch Substitution des Robinson-Algorithmus durch den neuen Unifikationsalgorithmus im Algorithmus $\mathcal{W}$.

Definition 3.16. Sei O eine Überladungsannahme. Dann sind die Funktionen $m : F \to \mathbf{2^{Id}}$ und $d : F \times \mathbb{N} \times \mathbf{2^{Id}} \to \mathbf{2^{Id}}$ wie folgt definiert:

$$\begin{aligned} m(f) &\stackrel{\text{def}}{=} \{x \mid x : \langle \omega, s\rangle \in O \wedge \exists \overline{\alpha_n}.\ f(\overline{\alpha_n}) \in s\} \\ d_f(i,X) &\stackrel{\text{def}}{=} \bigcup_{x \in X} \{ops(\alpha_i) \mid x : \langle \omega, s\rangle \in O \wedge \exists \overline{\alpha_n}.\ f(\overline{\alpha_n}) \in s\} \end{aligned}$$

Die Funktion m bildet Typkonstruktoren f auf die Menge von Operatoren ab, die Werte von Typen der Form $f(\overline{\tau_n})$ an überladenen Argumentpositionen akzeptieren. Die Funktion $d_f(i,X)$ bildet eine Menge von Typnamen ab auf die Menge Y_i, sodaß, falls $Y_i \uparrow^o \tau_i$ für $i = 1..n$ gilt und $X \subseteq m(f)$, dann gilt auch $X \uparrow^o f(\tau_1, \ldots, \tau_n)$.

Proposition 3.17. *Seien m und d wie oben definiert, dann gilt:*

$$X \uparrow^0 \tau \iff \begin{cases} \tau = \alpha \wedge X \subseteq ops(\alpha) \quad \textit{oder} \\ \tau = f(\tau_1, \ldots, \tau_n) \wedge X \subseteq m(f) \wedge \forall i = 1..n : d_f(i,X) \uparrow^0 \tau_i. \end{cases}$$

Sei S eine Substitution und O eine Überladungsannahme. S *respektiert* O, falls $\alpha_X \in dom(S) \Rightarrow X \uparrow S\alpha$. Für Typausdrücke τ_1 und τ_2 definieren wir: τ_2 ist eine gültige Instanz von τ_1 ($\tau_1 \geq^o \tau_2$), falls eine Substitution S existiert, die O respektiert, sodaß $\tau_2 = S\tau_1$. Für Substitutionen S_1 und S_2 gilt: S_1 ist allgemeiner als $S_2 (S_1 \geq^o S_2)$, falls $dom(S_1) \subseteq dom(S_2)$ und $\alpha \in dom(S_1) \Rightarrow S_1(\alpha) \geq^o S_2(\alpha)$.

Proposition 3.18. *Die Relation $\uparrow^o$ ist abgeschlossen unter Substitutionen: Falls $X \uparrow^o \tau$ gilt und S eine überladungssortierte Substitution ist, dann gilt auch $X \uparrow^o S(\tau)$.*

Beweis. Durch Induktion über τ:

Für $\tau = \alpha_Y$ gilt $X \uparrow^o \alpha_Y \iff X \subseteq Y$. Da S wohlsortiert ist, gilt auch $Y \uparrow^o S(\alpha_Y)$ und da $X \subseteq Y$ auch $X \uparrow^o S(\alpha_Y)$.

Sei $\tau = f(\overline{\tau_n})$. Aus $X \uparrow^o \tau$ folgt $X \subseteq m(f) \wedge \forall i = 1..n.\ d_f(i, X) \uparrow^o \tau_i$. Nach Induktionshypothese gilt aber auch $d_f(i, X) \uparrow^o S(\tau_i)$ und damit $X \uparrow^o S(f(\overline{\tau_n}))$. □

Korrollar 3.19. *Die Komposition $S_1 \circ S_2$ zweier überladungssortierter Substitutionen S_1, S_2 ist ebenfalls eine überladungssortierte Substitution.*

Beweis. Folgt direkt aus Proposition 3.18. □

Lemma 3.20 (Allgemeinste, Sorten respektierende Substitutionen). *Sei O eine Überladungsannahme, τ ein Typausdruck und X eine Menge von Operatornamen. Dann gibt es entweder keine Substitution die $X \uparrow^o S(\tau)$ erfüllt oder eine allgemeinste. Darüberhinaus gibt es einen effektiven Algorithmus cs_O zur Berechnung einer solchen Substitution:*

$$
\begin{array}{ll}
cs_O(X, \alpha) = \{\alpha \mapsto \beta\} & \beta \text{ eine neue Typvariable, mit } ops(\beta) = X \cup ops(\alpha) \\
cs_O(X, f(\tau_1, \ldots, \tau_n)) = S_n & \text{falls } X \subseteq m(f) \wedge \exists S_0 \ldots S_n \text{ sodaß} \\
 & \quad S_0 = \emptyset \wedge \forall i = 1..n : \\
 & \quad\quad S_i = cs_O(d_f(i, X), S_{i-1}(\tau_i)) \circ S_{i-1}
\end{array}
$$

In allen anderen Fällen schlägt $cs_O(X, \tau)$ fehl.

Beweis. Wir zeigen zunächst durch Berechnungsinduktion, daß, falls $cs_O(X, \tau)$ nicht fehlschlägt, eine Substitution S berechnet wird, die O respektiert und darüberhinaus $X \uparrow^o \tau$ erfüllt.

Zur Induktionsverankerung sei $\tau = \alpha$. Dann ist $S = \{\alpha \mapsto \beta\}$ und β eine neue Typvariable, die $ops(\beta) = X \cup ops(\alpha)$ erfüllt. Nun gilt $X \uparrow^o S(\alpha)$ da $X \subseteq ops(\beta)$ und S die Überladungsannahme O respektiert, da $S\alpha = \beta, ops(\alpha) \subseteq ops(\beta)$ und $ops(\alpha) \uparrow^o \beta$.

Für den Induktionsschritt $f(\overline{\tau_n})$ nehmen wir an, daß $X \subseteq m(f)$ und Substitutionen $S_0 \ldots S_n$ existieren, sodaß $S = S_n, S_0 = \emptyset \wedge \forall i = 1..n : S_i = cs_O(d_f(i, X), S_{i-1}(\tau_i)) \circ S_{i-1}$. S respektiert O, da S die Verknüpfung Sorten respektierender Substitutionen ist (Korrolar 3.19). Wir müssen nun zeigen, daß $X \uparrow^o S(f(\tau_1, \ldots, \tau_n))$, oder äquivalent: $X \uparrow^o f(S_n\tau_1, \ldots, S_n\tau_n))$, was wiederum äquivalent ist zu $X \subseteq m(f) \wedge \forall i = 1..n :$

$d_f(i, X) \uparrow^o S_n(\tau_i)$. Für $i \in 0..n$, kann S als $S'_i \circ S_i$ geschrieben werden, für eine Sorten respektierende Substitution S'_i. Da $X \subseteq m(f)$ und $d_f(i, X) \uparrow^o S_i(\tau_i)$ aufgrund der Induktionshypothese gilt, folgt $d_f(i, X) \uparrow^o S'_i(S_i(\tau_i))$ aufgrund von Proposition 3.18.

Es bleibt zu zeigen, daß für jede andere Substitution R, die $X \uparrow^o R\tau$ erfüllt, eine Substitution S' gefunden werden kann, sodaß $R = S' \circ S$. Der Beweis erfolgt wiederum durch Berechnungsinduktion:

Für $\tau = \alpha$ wählen wir $S' = R$, außer, daß $S'(\beta) = R(\alpha)$. Dann gilt $S'(S(\alpha)) = S'(\beta) = R(\alpha)$ wie verlangt. Für den Induktionsschritt sei $\tau = f(\tau_1, \ldots, \tau_n)$. Um zu zeigen, daß $R = S' \circ S$, zeigen wir, daß für jedes $i \in 0..n$ eine Substitution S'_i existiert, sodaß $R = S'_i \circ S_i$:

„$i = 0$“ Da $S_0 = \emptyset$ können wir $S'_0 = R$ wählen..

„$i - 1 \Rightarrow i$“ Nach Induktionsannahme gilt $R = S'_{i-1} \circ S_{i-1}$. Sei $\tau'_i = S_{i-1}(\tau_i)$, dann gilt $d_f(i, X) \uparrow^o S'_{i-1}(\tau'_i)$ und wir können die äußere Induktionshypothese anwenden, um $S'_{i-1} = V \circ cs_O(d_f(i, X), \tau'_i)$, zu erhalten, für eine Substitution V. Dies impliziert aber $R = S'_{i-1} \circ S_{i-1} = (V \circ cs(d_f(i, X), \tau'_i)) \circ S_{i-1} = V \circ S_i$. □

Mit Hilfe der Funktion cs_O definieren wir nun einen Unifikationsalgorithmus ähnlich dem Robinson-Algorithmus.[6]

Satz 3.21 (Allgemeinste Unifikatoren für überladungssortierte Terme). *Gegeben eine Überladungsannahme O, zwei überladungssortierte Terme τ_1 und τ_2, gibt es entweder keinen Unifikator S, sodaß $S(\tau_1) = S(\tau_2)$, oder einen allgemeinsten. Der in Abbildung 3.6 angegebene Algorithmus $\mathcal{U}_O$ bestimmt korrekt, ob es einen allgemeinsten Unifikator gibt und berechnet diesen.*

Beweis. Berechnungsinduktion analog zu Lemma 3.20. Lediglich der Fall $U_O(\alpha_x, \tau)$ ist interessant: enthält τ die Variable α_x, dann kann es keine Lösung des Unifikationsproblems geben. Für $\alpha_x \notin \mathcal{V}(\tau)$ müssen die Sorten der Variablen in τ so angepaßt

[6] Die hier vorgestellte Robinson-Variante hat natürlich ebenso wie der Original-Algorithmus das Problem der möglicherweise exponentiell wachsenden Terme (siehe Seite 14). In Abschnitt 3.4.3 geben wir eine effiziente Implementierung an, die dieses Problem vermeidet und zusätzlich auch reguläre Terme behandeln kann.

$$
\begin{array}{ll}
\mathcal{U}_O(\tau,\tau) = \emptyset & \\
\mathcal{U}_O(\tau,\alpha_x) = \mathcal{U}_O(\alpha_x,\tau) & \text{falls } \tau \notin \mathcal{V} \\
\mathcal{U}_O(\alpha_x,\tau) = \{\alpha_x \mapsto S(\tau)\} \circ S & \text{falls } S = cs_O(x,\tau) \text{ existiert, und } \alpha_x \notin \mathcal{V}(\tau) \\
\mathcal{U}_O(f(\overline{\tau_n}), f(\overline{\tau'_n})) = S_n & \text{falls } \exists S_0,\ldots,S_n : \\
 & \quad S_0 = \emptyset \wedge \forall i = 1..n : \\
 & \quad\quad S_i = \mathcal{U}_O(S_{i-1}(\tau_i), S_{i-1}(\tau'_i)) \circ S_{i-1}
\end{array}
$$

$\mathcal{U}_O(\tau,\tau')$ schlägt in allen anderen Fällen fehl.

Abbildung 3.6: Unifikation für überladungssortierte Terme

werden, daß $X \uparrow^o \tau$ erfüllt ist, was aufgrund von Lemma 3.20 durch Anwenden der Substitution $S = cs_O(X,\tau)$ genau dann sichergestellt werden kann, wenn diese existiert. □

3.4.1 Überladungssortierung vs. Ordnungssortierung

Wir werden nun zeigen, daß zwischen der Unifikation von überladungssortierten Termen und Unifikation für ordnungssortierte Termalgebren mit leerer Gleichungstheorie eine enge Verbindung existiert. Dazu führen wir zunächst die notwendigen Begriffe aus der Unifikationstheorie ein, und zeigen dann, wie man zu einer gegebenen parametrischen Überladungsannahme eine ordnungssortierte Termalgebra konstruieren kann, sodaß die entsprechenden Unifikationsbegriffe zusammenfallen.[7]

Eine *ordnungssortierte Signatur* ist ein Tripel $(S,\leq,\Sigma)$, wobei S eine Menge von Sorten, $\leq$ eine partielle Ordnung auf S und Σ eine Familie $\{\Sigma_{w,s} \mid w \in S^*, s \in S\}$ von nicht notwendigerweise disjunkten Operatorsymbolen ist. Zur Vereinfachung nehmen wir im folgenden an, daß sowohl S als auch Σ endlich sind. Statt $f \in \Sigma_{w,s}$ schreiben wir vereinfachend $f : w \to s$. $w \to s$ bezeichnen wir als die *Stelligkeit* des Operatorsymbols f. Im Unterschied zu freien Termalgebren erlauben wir mehr als eine

[7] Die Einführung der Begriffe folgt im wesentlichen der Darstellung in [NS91]. Dort wird die Verbindung zwischen dem Typsystem der Programmiersprache Haskell und ordnungssortierter Unifikation untersucht.

Stelligkeitsdeklaration für Operatorsymbole. In diesem Fall nennen wir das Operatorsymbol *überladen*. Die Ordnung $\leq$ wird auf kanonische Weise auf S^* fortgesetzt: $\overline{w_n} \leq \overline{w'_m} \iff m = n \wedge \forall i = 1..n : w_i \leq w'_i$. Sei $V = \{V_s \mid s \in S\}$ eine *S-sortierte* Familie von abzählbar unendlichen Menge von Variablen. Für $x \in V_s$ schreiben wir auch $x : s$ bzw. x_s.

Die Menge der über V frei generierten *ordnungssortierten* Terme $T_\Sigma(V)_s$ der Sorte s ist induktiv wie folgt definiert:

$$\frac{x \in V_s \quad s \leq s'}{x \in T_\Sigma(V)_{s'}} \qquad \frac{f : \overline{s_n} \to s \quad s \leq s' \quad \forall i.t_i \in T_\Sigma(V)_{s_i}}{f(\overline{t_n}) \in T_\Sigma(V)_{s'}}$$

Die Menge aller über V frei generierten, ordnungssortierten Terme $T_\Sigma(V)$ ist gegeben durch $T_\Sigma(V) = \bigcup_{s \in S} T_\Sigma(V)_s$. Man beachte, daß aufgrund dieser Definition Terme korrekt sortiert sein müssen: jeder Subterm eines zusammengesetzten Terms muß eine Sorte haben, die durch die Stelligkeit des Operators vorgegeben ist.

Ein Term kann mehr als eine Sorte haben: einerseits hat trivialerweise ein Term der Sorte s auch alle Sorten $s' \geq s$, andererseits kann ein Term aufgrund der Überladung von Operatoren $f : w \to s$ sowie $f : w \to s'$ zwei möglicherweise unvergleichbare Sorten besitzen. Um solche Anomalien auszuschließen, arbeitet man meistens mit regulären Signaturen: eine Signatur heißt *regulär*, falls jeder Term eine kleinste Sorte besitzt. Regularität ist entscheidbar:

Satz 3.22. *[SNGM89] Eine Signatur $(S, \leq, \Sigma)$ ist regulär, genau dann, wenn für jedes $f \in \Sigma$ und $w \in S^*$ die Menge $\{s \mid \exists w' \geq w.f : w' \to s\}$ entweder leer ist, oder ein kleinstes Element besitzt.*

Substitutionen $\theta \in V \to T_\Sigma(V)$ sind in der üblichen Weise definiert. Vergleichbar mit dem Vorgehen für überladungssortierte Terme, verlangt man jedoch zusätzlich, daß Substitutionen *sorten-korrekt* sind: $\theta(x_s) \in T_\Sigma(V)_s$. Die Komposition zweier sorten-korrekter Substitutionen ist ebenfalls sorten-korrekt.

Eine Unifikator θ einer Menge M von Gleichungen über ordnungssortierten Termen ist eine sorten-korrekte Substitution mit der Eigenschaft $\theta s = \theta t$ für alle $s = t \in M$. Eine Menge von Unifikatoren U heißt *vollständig*, falls für jeden Unifikator θ' von M ein $\theta \in U$ existiert, sodaß $\theta' \leq \theta$; U heißt minimal, falls $\theta \not\leq \theta'$ für alle $\theta, \theta' \in U$. Im Unterschied zur Robinson-Unifikation auf unsortierten Termen, kann es jedoch

sein, daß für eine gegebene Signatur, vollständige, minimale Mengen von Unifikatoren mehr als ein Element enthalten. Man nennt daher eine Signatur *unitär (unifizierend)*, falls für jede Menge M von Termgleichungen eine vollständige, minimale Menge von Unifikatoren $\mu_\Sigma(M)$ existiert, die höchstens ein Element enthält. Eine Signatur heißt *finitär (unifizierend)*, falls $\mu_\Sigma(M)$ höchstens endlich viele Elemente enthält.

Für nicht reguläre Signaturen kann Unifikation infinitär sein, sogar bei endlicher Signatur. Es gilt jedoch:

Satz 3.23. *[SS85] Für endliche und reguläre Signaturen haben endliche Mengen von Termgleichungen endliche, vollständige und effektiv berechenbare Mengen von Unifikatoren.*

Für unsere Zwecke erwarten wir jedoch eine unitär unifizierende Signatur. Unitäre Signaturen können u.a. wie folgt charakterisiert werden: Eine Signatur heißt *abwärts vollständig*, falls zwei Sorten entweder keine untere Schranke besitzen oder aber eine größte. Eine Signatur heißt *co-regulär*, falls für jeden Operator f und jede Sorte s, die Menge $D(f,s) = \{w \mid \exists s'.f : w \to s' \wedge s' \leq s\}$ entweder leer ist, oder ein größtes Element hat.

Satz 3.24. *[SNGM89] Jede endliche reguläre, co-reguläre und abwärts vollständige Signatur ist unitär unifizierend.*

Nach diesen Vorbereitungen können wir nun daran gehen, zu einer gegebenen Überladungsannahme O eine unitäre, ordnungssortierte Termalgebra zu konstruieren.

Definition 3.25. Sei O eine Überladungsannahme. Dann konstruieren wir die Signatur $\Sigma_O = (S, \leq, \Sigma)$ wie folgt:

$$
\begin{aligned}
&S = \{m \mid m \subseteq dom(O)\} \\
&s \leq s' \iff s \supseteq s' \\
&\Sigma = \{f : d_f(1,y), \ldots, d_f(n,y) \to y \mid y \subseteq m(f)\}
\end{aligned}
$$

Proposition 3.26. $\tau \in T_F(V) \iff \tau \in T_{\Sigma_O}(V)$

Beweis. Folgt direkt aus der Tatsache, daß jeder beliebige Term in $T_F(V)$ sortenkorrekt in $T_{\Sigma_O}(V)$ ist, da jeder Operator $f \in F$ mit Stelligkeit $\rho(f) = n$ in Σ_O die Stelligkeit $f : \emptyset^n \to \emptyset$ besitzt und $\tau : s \Rightarrow \tau : s'$ für alle $s' \geq s$. □

Proposition 3.27. *Sei $X \subseteq dom(O)$, dann gilt $X \uparrow^o \tau \iff \tau : X$.*

Beweis. Durch strukturelle Induktion über τ.

Für $\tau = \alpha_Y \in V$ gilt:

$$\begin{aligned} X \uparrow^o \alpha_Y &\iff X \subseteq Y \\ &\iff Y \leq X \\ &\iff \alpha_Y : X \end{aligned}$$

Sei $\tau = f(\overline{\tau_n})$, dann gilt:

$$\begin{aligned} X \uparrow^o f(\overline{\tau_n}) &\iff X \subseteq m(f) \wedge \forall i = 1..n.\ d_f(i, X) \uparrow^o \tau_i \\ &\iff X \subseteq m(f) \wedge \forall i = 1..n.\ \tau_i : d_f(i, X) \\ &\iff f : d_f(1, X), \ldots, d_f(n, X) \to X \in \Sigma \wedge \forall i = 1..n.\ \tau_i : d_f(i, X) \\ &\iff f(\overline{\tau_n}) : X \end{aligned}$$ □

Daraus folgt sofort, daß jede Substitution, welche die Überladungsannahme O respektiert, auch eine wohlsortierte Substitution gemäß Σ_O ist und umgekehrt. Daher gilt:

Korrollar 3.28. *Jeder O-respektierende Unifikator ist auch ein Unifikator in T_{Σ_O}*

Beweis. Folgt aus den Propositionen 3.26 und 3.27. □

Wie in Abschnitt 3.4 gezeigt, ist Unifikation für überladungssortierte Terme unitär, somit muß aufgrund von Korrollar 3.28 auch Σ_O unitär unifizierend sein. Wir zeigen jedoch direkt:

Satz 3.29. *Sei O eine Überladungsannahme und Σ_O konstruiert gemäß Definition 3.25. Dann ist Σ_O endlich, regulär, co-regulär und abwärts vollständig und somit unitär unifizierend.*

Beweis. Die *Endlichkeit* der Signatur ist offensichtlich.

Zum Beweis der *Regularität* betrachten wir aufgrund von Satz 3.22 für $f \in \Sigma$, $w \in S^*$ die Menge $M_w^f = \{s \mid \exists w' \geq w.f : w' \to s\}$. Wir zeigen, daß aus $s_1, s_2 \in M_w^f$ auch

$s_1 \sqcap s_2 \in M_w^f$ folgt. Da M_w^f endlich ist, enthält dann auch jede nicht leere Menge M_w^f ein kleinstes Element.

Sei also $s_1, s_2 \in M_w^f$. Dann gilt $s_1 \sqcap s_2 \in M_w^f \iff \exists w' \geq w.f : w' \to s_1 \sqcap s_2$. Nach Konstruktion von Σ_O gilt aber mit $f : w_1 \to s_1 \in M_w^f$ und $f : w_2 \to s_2 \in M_w^f$ auch $f : w_1 \sqcap w_2 \to s_1 \sqcap s_2 \in M_w^f$, da $f : w_1 \sqcap w_2 \to s_1 \sqcap s_2$ und aus $w_1 \geq w$ und $w_2 \geq w$ auch $s_1 \sqcap s_2 \geq w$.[8]

Zum Beweis der *Co-Regularität* betrachten wir für beliebige $f \in \Sigma$, $s \in S$ die Menge $D(f,s) = \{w \mid \exists s'.f : w \to s' \wedge s' \leq s\}$. Analog zur Regularität zeigen wir daß aus $w_1, w_2 \in D(f,s)$ auch $w_1 \sqcup w_2 \in D(f,s)$ folgt. $w_1 \in D(f,s) \Rightarrow \exists s_1 \leq s.f : w_1 \to s_1$, $w_2 \in D(f,s) \Rightarrow \exists s_2 \leq s.f : w_2 \to s_2$. Nach Konstruktion von Σ_O gilt $f : w_1 \sqcup w_2 \to s_1 \sqcup s_2$. Da $s_1 \sqcup s_2 \leq s$ ist also auch $w_1 \sqcup w_2 \in D(f,s)$, sodaß aus der Endlichkeit von $D(f,s)$ die Existenz eines maximalen Elements in $D(f,s)$ die Regularität von Σ_O folgt.

Die Menge S bildet zusammen mit der definierten Relation $\leq$ einen vollständigen Verband, daher ist die Signatur auch *abwärts vollständig*: zu je zwei Sorten s_1, s_2 enthält die Menge der maximalen Elemente unterhalb von s_1 und s_2 genau ein Element.

$$\max_{\leq}\{s \mid s \leq s_1 \wedge s \leq s_2\} = \min_{\supseteq}\{s \mid s \supseteq s_1 \wedge s \supseteq s_2\} = \{s_1 \cup s_2\} \qquad \square$$

3.4.2 Konstruktorvariablen

Im Folgenden betrachten wir eine Erweiterung der Syntax der Überladungsschemata, die der Autor schon in der ersten Veröffentlichung zum Thema parametrische Überladungen erwähnt hat [Kae88], aber bisher nicht formal behandelt hat, weil diese Erweiterung zur damaligen Zeit nur unwesentlich mehr Überladungsmöglichkeiten zu erlauben schien.

Als motivierendes Beispiel betrachten wir eine Operation *contains*, die auf das Vorhandensein eines Elementes in einer Datenstruktur mit einem bestimmten Elementtyp testen soll. D.h. wir suchen ein Überladungsschema ω, das die folgenden Instan-

[8] Dabei werden die Relation $\supseteq$ und der Operator $\sqcap$ etc. auf kanonische Weise auf S^* erweitert: $\overline{w_n} \sqcap \overline{w'_n} = w_1 \sqcap w'_1, \ldots, w_n \sqcap w_n$.

zen zuläßt:

$$\forall\alpha_{\{=\}}.\ \alpha_{\{=\}} \to \mathit{list}(\alpha_{\{=\}}) \to \mathit{bool}$$
$$\forall\alpha_{\{=\}}.\ \alpha_{\{=\}} \to \mathit{set}(\alpha_{\{=\}}) \to \mathit{bool}$$
$$\forall\alpha_{\{=\}}.\ \alpha_{\{=\}} \to \mathit{tree}(\alpha_{\{=\}}) \to \mathit{bool},\ \ldots$$

Offensichtlich ist $\$ \to \$ \to \mathit{bool}$ nicht korrekt, da beide Argumente mit verschiedenen Typen instanziert werden müßten. Und doch sieht man, daß im Prinzip eine einzige Variable alle Überladungsinstanzen beschreiben kann, da sich die Instanzen nur im äußeren Typkonstruktor des zweiten Argumentes unterscheiden.

Wir erweitern daher die Menge der zulässigen Überladungsschemata und erlauben die Verwendung des Symbols \$ an der Stelle von Typkonstruktoren. Dabei darf \$ mehrmals verwendet werden, jedoch jeweils mit den gleichen Typausdrücken als Argument. Dies läßt sich wie folgt formalisieren:

Definition 3.30 (Termpositionen, τ/p). Für $\tau \in T_F(\mathcal{V})$ sei die Menge der Termpositionen $\mathit{Pos}(\tau) \subseteq \mathbb{N}^*$ definiert durch

$$\mathit{Pos}(\alpha) = \{\epsilon\}$$
$$\mathit{Pos}(f(\overline{\tau_n})) = \{\epsilon\} \cup \{i{\cdot}p \mid i \in 1..n \wedge p \in \mathit{Pos}(\tau_i)\}$$

wobei ϵ die leere Sequenz und $i{\cdot}p$ die Konkatenation bezeichnet. Dann wird für $p \in \mathit{Pos}(\tau)$ der Subterm an der Position p durch τ/p notiert und ist wie folgt definiert:

$$\tau/\epsilon = \tau$$
$$f(\overline{\tau_n})/i{\cdot}p = \tau_i/p$$

Damit lassen sich die gültigen Überladungsschemata wir folgt definieren: ω ist gültig, falls für alle $p, q \in \mathit{Pos}(\omega)$ mit $\omega/p = \$(\overline{\tau_n})$ und $\omega/q = \$(\overline{\tau'_m})$ auch $\overline{\tau_n} = \overline{\tau'_m}$ gilt. Beispiel:

$$\mathit{contains} : \forall\alpha_{\{=\}}.\ \alpha_{\{=\}} \to \$(\alpha_{\{=\}}) \to \mathit{bool},\ \{\mathit{list}(\alpha), \mathit{set}(\alpha_{\{=\}}), \mathit{tree}(\alpha)\}$$

Zur Repräsentation der Typen solcherart überladener Operatoren, erweitern wir die Menge der Typvariablen um eine Familie abzählbarer Mengen von *Konstruktorvariablen*

$$\mathcal{V} = \bigcup_{X \subseteq dom(O)} \mathcal{V}_X^{\alpha} \quad \cup \quad \bigcup_{X \subseteq dom(O)} \mathcal{V}_X^{\kappa}$$

sowie die abstrakte Syntax der Typausdrücke

$$\tau ::= \alpha_X \mid f(\tau_1, \ldots, \tau_n) \mid \kappa_X(\tau_1, \ldots, \tau_n)$$

und redefinieren die Menge der zulässigen Argumente eines Operators $x \in dom(O)$ wie folgt:

Definition 3.31. τ ist ein zulässiger Argumenttypausdruck für x, geschrieben $x \uparrow^o \tau$, falls eine der folgenden Bedingungen gilt

$$\begin{array}{l}\tau = \alpha_x \wedge x \in X \\ \tau = \kappa_X(\overline{\tau_n}) \wedge x \in X \\ \tau = f(\overline{\tau_n}) \wedge O(x) = \langle \omega, s\rangle \wedge f(\overline{\alpha_n}) \in s \wedge \forall i = 1..n : \forall y \in ops(\alpha_i) : y \uparrow^o \tau_i\end{array}$$

Entsprechend Definition 3.4 legt man wiederum fest: $X \uparrow^o \tau \iff \forall x \in X.\ x \uparrow^o \tau$.

Überladungsschemata werden in Typannahmen durch Typschemata repräsentiert, indem man die \$-Symbole durch entsprechende Typvariablen aus $\mathcal{V}^\kappa$ ersetzt und diese abstrahiert. Damit erhält man für das obige Beispiel folgende Typannahme:

$$contains : \forall \kappa_{\{contains\}}.\forall \alpha_{\{=\}}.\ \alpha_{\{=\}} \to \kappa_{\{contains\}}(\alpha_{\{=\}}) \to bool,$$

Substitutionen sind wie üblich definiert, wobei Konstruktorvariablen nur auf Konstruktoren abgebildet werden, d.h. $S(\kappa) \in F$ für $\kappa \in dom(S) \cap \mathcal{V}^\kappa$ und $S(\alpha) \in T_F(\mathcal{V})$ für $\alpha \in dom(S) \cap \mathcal{V}^\alpha$. Jedoch muß man fordern, daß der Bildbereich einer Substitution *konstruktorvariablenkompatibel* ist.

Definition 3.32. Eine Menge Z von Typausdrücken ist *konstruktorvariablenkompatibel*, falls die folgende Bedingung erfüllt ist:

$$\begin{array}{r}\forall \tau_1, \tau_2 \in Z : \forall p \neq q \in Pos(\tau) \text{ wobei } \tau = f(\tau_1, \tau_2) \text{ für ein } f \in F_2 : \\ \tau/p = \kappa(\overline{\tau_n}) \wedge \tau/q = \kappa(\overline{\tau'_m}) \Rightarrow \overline{\tau_n} = \overline{\tau'_m} \qquad \text{(kvk)}\end{array}$$

Proposition 3.33. *Sei τ konstruktorvariablenkompatibel (d.h. $\{\tau\}$ erfüllt (kvk)) und S eine Substitution, deren Bildbereich $rng(S)$ Bedingung (kvk) erfüllt, dann erfüllt auch $S(\tau)$ Bedingung (kvk).*

$$
\begin{array}{ll}
cs_o^\kappa(X,\alpha) = \{\alpha \mapsto \beta\} & \beta \text{ eine neue Typvariable} \\
& \quad \text{mit } ops(\beta) = X \cup ops(\alpha) \\
cs_o^\kappa(X, f(\overline{\tau_n})) = S_n & \text{falls } X \subseteq m(f) \wedge \exists S_0 \ldots S_n \text{ sodaß} \\
& \quad S_0 = \emptyset \wedge \forall i = 1..n : \\
& \quad\quad S_i = cs_O^\kappa(d_f(i,X), S_{i-1}(\tau_i)) \circ S_{i-1} \\
cs_o^\kappa(X, \kappa(\overline{\tau_n})) = \{\kappa \mapsto \kappa'\} & \kappa' \text{ eine neue Typvariable} \\
& \quad \text{mit } ops(\kappa') = X \cup ops(\kappa) \\
\text{In allen anderen Fällen schlägt } cs_o^\kappa(X,\tau) \text{ fehl.} &
\end{array}
$$

Abbildung 3.7: Sortenanpassung für Terme mit Konstruktorvariablen

Daraus folgt auch, daß die Verknüpfung zweier Sorten respektierender, konstruktorvariablenkompatibler Substitutionen wiederum eine Sorten respektierende, konstruktorvariablenkompatible Substitution ist. Einfache und generische Instanz auf Typausdrücken und Substitutionen sind wie oben definiert, nur daß nun auch über Konstruktorvariablen abstrahiert werden darf. Man überzeugt sich leicht, daß damit sämtliche interessanten syntaktischen Eigenschaften der parametrischen Überladungen erhalten bleiben.

Grundlage eines erweiterten Unifikationsalgorithmus ist auch hier ein Algorithmus zur Berechnung allgemeinster Sorten respektierender Substitutionen.

Lemma 3.34 (Sortenanpassung für Terme mit Konstruktorvariablen). *Sei O eine Überladungsannahme, τ ein Typausdruck und X eine Menge von Operatornamen. Dann gibt es entweder keine Substitution die $X \uparrow^o S(\tau)$ erfüllt oder eine allgemeinste. Darüberhinaus gibt es einen effektiven Algorithmus cs_o^κ zur Berechnung einer solchen Substitution.*

Algorithmus cs_o^κ ist eine konservative Erweiterung des Algorithmus cs für einfache überladungssortierte Terme und ist in Abbildung 3.7 angegeben.

Beweis. (von Lemma 3.34) Analog zu Lemma 3.20. Lediglich der Fall $\tau = \kappa(\overline{\tau_n})$ kommt neu hinzu und ist symmetrisch zu dem Fall $\tau = \alpha$. □

$\mathcal{U}_o^\kappa(\tau,\tau) = \emptyset$

$\mathcal{U}_o^\kappa(\tau,\alpha_x) = \mathcal{U}_o^\kappa(\alpha_x,\tau)$ falls $\tau \notin \mathcal{V}$

$\mathcal{U}_o^\kappa(\alpha_x,\tau) = \{\alpha_x \mapsto S(\tau)\} \circ S$ falls $S = cs_o^\kappa(x,\tau)$ existiert, und $\alpha_x \notin \mathcal{V}(\tau)$

$\mathcal{U}_o^\kappa(f(\overline{\tau_n}), f(\overline{\tau'_n})) = S_n$ falls $\exists S_0,\ldots,S_n.$
$S_0 = \emptyset \wedge \forall i = 1..n :$
$S_i = \mathcal{U}_o^\kappa(S_{i-1}(\tau_i), S_{i-1}(\tau'_i)) \circ S_{i-1}$

$\mathcal{U}_o^\kappa(f(\overline{\tau_n}), \kappa(\overline{\tau'_n})) = \mathcal{U}_o^\kappa(\kappa(\overline{\tau'_n}), f(\overline{\tau_n}))$

$\mathcal{U}_o^\kappa(\kappa(\overline{\tau_n}), f(\overline{\tau'_n})) = S'' \circ S' \circ S$ falls $\exists S, S', S''.$
$S = cs_o^\kappa(ops(\kappa), f(\overline{\tau'_n}))$
$S' = \mathcal{U}_o^\kappa(S(f(\overline{\tau_n})), S(f(\overline{\tau'_n})))$
$S'' = \{\kappa \mapsto f\}$

$\mathcal{U}_o^\kappa(\kappa_1(\overline{\tau_n}), \kappa_2(\overline{\tau'_n})) = S \circ S_n$ falls $\kappa_1 \neq \kappa_2$ und $\exists S_0,\ldots,S_n.$
$S_0 = \emptyset \wedge \forall i = 1..n :$
$S_i = \mathcal{U}_o^\kappa(S_{i-1}(\tau_i), S_{i-1}(\tau'_i)) \circ S_{i-1}$
wobei $S = \{\kappa_1 \mapsto \kappa,\ \kappa_2 \mapsto \kappa\}$
und κ eine neue Variable
mit $ops(\kappa) = ops(\kappa_1) \cup ops(\kappa_2)$

$\mathcal{U}_o^\kappa(\tau,\tau')$ schlägt in allen anderen Fällen fehl.

Abbildung 3.8: Unifikation für überladungssortierte Terme mit Konstruktorvariablen

Auf der Grundlage dieses Resultats definieren wir nun den Unifikationsalgorithmus $\mathcal{U}_o^\kappa$ für überladungssortierte Terme mit Konstruktorvariablen wie in Abbildung 3.8 angegeben.

Satz 3.35 (Allgemeinste Unifikatoren für überladungssortierte Terme mit Konstruktorvariablen). *Sei O eine Überladungsannahme und τ_1, τ_2 zwei überladungssortierte, konstruktorvariablenkompatible Terme, dann gibt es entweder keinen Unifikator S, sodaß $S(\tau_1) = S(\tau_2)$, oder einen allgemeinsten. Algorithmus $\mathcal{U}_o^\kappa$ bestimmt korrekt, ob es einen allgemeinsten Unifikator gibt und berechnet diesen.*

Beweisskizze. Die ersten vier Gleichungen entsprechen der „einfachen“ überla-

dungssortierten Unifikation. Gleichung 4 führt die Unifikation eines Argumentpaares der Form $f(\overline{\tau_n}), \kappa(\overline{\tau'_n})$ zurück auf Fall 5 unter Vertauschung der Argumente und ist wegen der Kommutativität der Unifikation offensichtlich korrekt. Somit verbleiben die Fälle 5 und 6.

Im Fall 5 müssen zur Bestimmung eines allgemeinsten Unifikators von $\kappa(\overline{\tau_n}), f(\overline{\tau'_n})$ die Sorten angepaßt werden (1. Zeile), die Teilterme unifiziert werden (2. Zeile) und schließlich κ durch f ersetzt werden (3. Zeile). Offensichtlich ist $S'' \circ S' \circ S$ ein Unifikator, daß er auch der allgemeinste ist, folgt aus Lemma 3.34.

Im Fall 6 (Argumentpaar $\kappa_1(\overline{\tau_n}), \kappa_2(\overline{\tau'_n})$) werden die Teilterme unifiziert und die beiden Konstruktorvariable durch eine neue Variable ersetzt, die mit allen Operatoren von κ_1 und κ_2 markiert sind. $S \circ S_n$ ist ein Unifikator für das Argumentpaar, daß er der allgemeinste ist, folgt aus der Sortierung von κ.

Aufgrund der Forderung, daß alle Konstruktorvariablenteilterme mit identischen Konstruktorvariablen identisch sind, kann der Fall $\kappa(\overline{\tau_n}), \kappa(\overline{\tau'_m})$ mit $\overline{\tau_n} \neq \overline{\tau'_m}$ nicht auftreten. Man überzeugt sich leicht, daß die von $\mathcal{U}_o^\kappa$ gelieferten Substitutionen diese Bedingung erhalten. □

Um zu sehen, warum die etwas unnatürlich erscheinende Forderung (kvk) an Konstruktorteilterme notwendig ist, betrachten wir das Beispiel

$$\mathcal{U}_o^\kappa(\, f(\kappa(\alpha), \kappa(\beta)),\ f(\gamma_{\{=\}}, \mathit{list}(\delta))\,)$$

unter einem erweiterten Algorithmus, mit der zusätzlichen, naheliegenden Gleichung

$$\mathcal{U}_o^\kappa(\kappa(\overline{\tau_n}), \kappa(\overline{\tau'_n})) = \mathcal{U}_o^\kappa(f(\overline{\tau_n}), f(\overline{\tau'_n})) \quad \text{für ein } f \in F_n,$$

die ungleiche Konstruktorteilterme zuläßt und durch rekursive Anwendung von $\mathcal{U}_o^\kappa$ zu unifizieren versucht. Durch manuelle Ausführung des Algorithmus erhält man die Substitution

$$S = \{\ \kappa_{\{=\}} \mapsto \mathit{list},\ \delta \mapsto \delta_{\{=\}},\ \beta \mapsto \delta_{\{=\}},\ \gamma_{\{=\}} \mapsto \mathit{list}(\alpha),\ \kappa \mapsto \mathit{list}\ \}$$

Durch Anwenden auf die beiden Argumente erhält man

$$S(f(\kappa(\alpha), \kappa(\beta))) = f(\mathit{list}(\alpha), \mathit{list}(\delta_{\{=\}})) = S(f(\gamma_{\{=\}}, \mathit{list}(\delta)))$$

Man sieht, daß S zwar ein Unifikator ist, aber zu allgemein, da α nicht, wie eigentlich erforderlich, durch eine neue Variable $\alpha'_{\{=\}}$ ersetzt wurde.

Bedauerlicherweise ist die Forderung, daß alle Teilterme eines Überladungsschemas, die mit \$ konstruiert sind, identisch sein müssen, sehr einschränkend, sodaß sich z.B. die Monaden-Operationen [Mog89, Wad90, Wad92] damit nicht behandeln lassen.[9] Monaden spielen u.a. eine wichtige Rolle bei der Modellierung von Ein/Ausgabeoperationen in rein funktionalen Programmiersprachen und daher wäre es wünschenswert, sie behandeln zu können, ähnlich dem Vorgehen in [Jon93].

Man könnte den Algorithmus $\mathcal{U}_o^\kappa$ so abändern, daß alle Kunstruktorvariablenteilterme in einer separaten Datenstruktur gespeichert und bei jeder Substitution einer Konstruktorvariablen die korrekte Anpassung der Sorten aller Teilterme über die Funktion cs_o^κ geprüft und berechnet wird. Es gibt jedoch keine Handhabe, diese Teilterme in den Typschemata für let-gebundene Bezeichner darzustellen. Dies wird erst durch den Übergang zu einem Typsystem mit eingeschränkten Typen möglich (siehe Kapitel 5). Wir verzichten daher auf die weitere Ausführung.

3.4.3 Implementierung der Überladungsunifikation

Es ist wohlbekannt, daß bei der Robinson-Variante der Unifikationsimplementierung die Größe der unifizierten Terme exponentiell wachsen kann und damit auch zu exponentiellen Laufzeiten führen kann. Wir geben hier einen polynomial zeitbeschränkten Algorithmus an, der die Überladungsunifikation mit Hilfe von „Structure-Sharing“ [BM72] ähnlich [CB83] implementiert. Diese Version wird auch im SAMPλE-System [GKT90] eingesetzt.

Grundlage der Implementierung ist die Darstellung der Terme als gerichtete azyklische Graphen (DAG). Die Datenstrukturen und die elementaren Operationen sind in Abbildung 3.9 angegeben, der eigentliche Unifikationsalgorithmus in Abbildung 3.10. Die Knoten des Graphen werden durch Objekte auf dem Heap (Typdeklaration term) und die Kanten durch Zeiger dargestellt (Typ tptr). Zur Vereinfachung wurde auf die Detaillierung der Speicherung von Term-Listen und Operatormengen verzichtet

[9] Der Operator *bind* würde das Überladungsschema $\$(\alpha) \to (\alpha \to \$(\beta)) \to \$(\beta)$ verlangen, der Operator *join* das Schema $\$(\$(\alpha)) \to \$(\alpha)$.

```
1  type
2     tptr = ^term;
3     term = record
4               kind: enum(vari, ctor);
5               name: integer;
6               sort: set (1 .. k);
7               sons: list (tptr);
8               set : tptr;
9               size: integer;
10              mark: integer;
11            end;
12 var
13    mark: integer; {initially 0}
14
15 union(s:term, t:term) ≡
16       if (s^.size>t^.size) then
17         union(t,s);
18       else begin
19         s^.set := t;
20         t^.size := t^.size + s^.size;
21       end
22
23 find(s:term) ≡
24       if s^.set ≠s then
25         s^.set := find(s^.set);
26       return s;

27 occurs(s:term, t:term) ≡
28       mark := mark+1;
29       return oc(s,t)
30
31 oc(s:term, t:term) ≡
32       t := find(t); { s.set=s }
33       if (t^.mark=mark) then
34         return false;
35       else begin
36         t^.mark := mark;
37         if (s=t) then
38           return true;
39         else if t^.kind = vari then
40           return false;
41         else begin
42           res := false;
43           l := t^.sons;
44           while ¬res do begin
45             res := oc(s,hd(l));
46             l := tl(l);
47           end;
48           return ok;
49         end
50
51 cs(s:set (1..k), t:term) ≡
52       t := find(t);
53       f := t^.name;
54       if (s ⊆ t^.sort) then
55         return true;
56       else if (t^.kind=vari) then
57         t^.sort := t^.sort ∪ s;
58       else if (s ⋤ m(f)) then
59         return false;
60       else begin
61         t^.sort:=t^.sort ∪ s;
62         l := t^.sons;  ok := true;
63         i := 1; k := length(l);
64         while ok ∧ i≤k do begin
65           ok := cs(d(f,i,s), hd(l));
66           i := i+1;
67           l := tl(l);
68         end
69         return ok;
70       end
71
```

Abbildung 3.9: Typdefinitionen und Hilfsfunktionen für Unify

```
72  unify(s:term, t:term) ≡
73      s, t := find(s), find(t);
74      if (s^.kind=vari ∧t^.kind=vari) then
75          unify(t,s);
76      else if (s^.kind=vari ∧t^.kind=ctor) then
77          if occurs(s,t) then
78              return false;
79          else
80          if (¬ cs(s.sort,t)) then
81              return false;
82          else begin
83              union(s,t);
84              return true;
85          end
86      else if (s^.name≠t^.name) then
87          return false;
88      else begin
89          union(s,t);
90          if (s.set=s) then
91            s.sort := s.sort ∪ t.sort;
92          else
93            t.sort := s.sort ∪ t.sort;
94          ok := true;
95          l1, l2:= s^.sons, t^.sons;
96          while ok do begin
97              ok := unify(hd(l1), hd(l2));
98              l1, l2:= tl(l1), tl(l2);
99          end
100         return ok;
101     end
```

Abbildung 3.10: Implementierung der überladungssortierten Unifikation

(Zeilen 6 und 7). Wir gehen jedoch davon aus, daß Konstruktor-, Operator- und Variablennamen durch ganze Zahlen repräsentiert werden.

Der Algorithmus *unify(s,t)* berechnet die feinste Äquivalenzrelation auf den Knoten des Termgraphen, die $s \equiv t$ umfaßt und kompatibel zur Termformation ist. Dabei werden Substitutionen im Gegensatz zum Robinson-Algorithmus nur implizit berechnet, Variablen werden in den Termen sofort durch Vereinigung der dem zu unifizierenden Teilterm und der Variable zugeordneten Äquivalenzklassen ersetzt. Die Vereinigung von Äquivalenzklassen und das Auffinden des Terms, der einer Äqui-

valenzklasse zugeordnet ist, erfolgt mit Hilfe des UNION-FIND Algorithmus (Felder *set* und *size* in Zeile 8 und 9, Funktionen *union*, Zeile 15, und *find*, Zeile 23).

Zur effizienten Implementierung des occur-checks wird eine globale Variable *mark* verwendet, die vor jeder Traversierung inkrementiert wird (Zl. 28) und mit deren Wert besuchte Knoten markiert werden (Zl. 36). Durch den Vergleich in Zeile 33 wird sichergestellt, daß bei einem Aufruf der Funktion *occurs* maximal $n \cdot e$ Aufrufe von *oc* erfolgen, wobei n die Anzahl der Knoten und e die Anzahl der Kanten des Termgraphen ist. Da die Anzahl der von einem Knoten ausgehenden Kanten durch $c_e = \max\{n \mid F_n \neq \emptyset\}$ begrenzt ist, gilt $e \leq c_e \cdot n$ und damit beträgt der Aufwand für einen occur-check $O(n)$. Da vor jedem rekursiven Aufruf von *unify* die Anzahl der Äquivalenzklassen um eine vermindert wird, terminiert der Algorithmus nach spätestens n Schritten, sodaß für die Aufrufe von *occurs* insgesamt $O(n^2)$ Zeitaufwand anfällt.

Die Funktion *cs* implementiert die Sortenanpassung von Variablen, unter Verwendung der bekannten Funktionen $m(f)$ und $d_f(i, X)$. Dabei wird ausgenutzt, daß jeder Teilterm mit der Menge von Operatoren markiert ist, deren Definiertheit für den entsprechenden Teilterm bereits nachgewiesen wurde. Für die Zeitabschätzung beobachten wir zunächst, daß vor jedem rekursiven Aufruf von *cs*, die Anzahl der Operatoren in der Operatorenmenge des aktuellen Teilterms mindestens um 1 erhöht wird. Daraus folgt, daß die Zeilen 56 bis 70 maximal $(n \cdot k)$-mal durchlaufen werden. Somit ergibt sich der Gesamtaufwand $T(cs)$ für alle Aufrufe von *cs* durch

$$T(cs) \leq c \cdot n \cdot k \cdot (T(Z_s) + T(Z_r))$$

wobei $T(Z_s)$ den Aufwand für die einmalige Ausführung von Zeilen 52 bis 55 und $T(Z_r)$ den Aufwand für die einmalige Ausführung von Zeilen 56 bis 70 bezeichnet. Wählt man zur Repräsentation von Operatormengen z.B. balancierte Bäume, dann kann der Aufwand für $s_1 \subseteq s_2$ und auch $s_1 \cup s_2$ mit $O(k \cdot \log k)$ begrenzt werden. Somit gilt

$$T(Z_s) \leq c_s \cdot k \cdot \log k\,.$$

Bei geeigneter Wahl ist $m(f)$ mit konstantem Aufwand realisierbar, die Funktion $d_f(i, X)$ kann mit k Vereinigungen von Operatormengen implementiert werden und ist somit von der Ordnung $O(k \cdot k \cdot \log k)$. Da die maximale Stelligkeit von Typkon-

struktoren durch eine Konstante beschränkt ist, ergibt sich daraus

$$T(cs) = O(n \cdot k^3 \cdot \log k)$$

Für die Operationen *union* und *find* gilt folgende Analyse: nach [AHU74, Theorem 4.4, Kapitel 4.7], wird jede Sequenz von $c \cdot n$ union- und find-Operationen in $c' \cdot n \cdot G(n)$ Schritten ausgeführt, wobei c' eine Konstante ist, die von c abhängt, und $G(n)$ die kleinste natürliche Zahl m ist, sodaß $F(m) \geq n$, wobei F durch $F(0) = 1, F(i+1) = 2^{F(i)}$ definiert ist.[10] *unify* führt $c \cdot n$ union-find-Operationen aus, *oc* weitere $c' \cdot n^2$ find-Operationen und *cs* schließlich $c'' \cdot n \cdot k$ find-Operationen. Damit läßt sich der Aufwand $T(uf)$ für die union-find-Operationen wie folgt bestimmen:

$$T(uf) = O(G(\max(n^2, n \cdot k)) \cdot \max(n^2, n \cdot k))$$

für den Algorithmus mit occur-check und

$$T(uf) = O(n \cdot k \cdot G(n \cdot k))$$

ohne occur-check.

Es ergibt sich folgende Zeitabschätzung für Gesamtalgorithmus:

$$T(unify) = O(n^2 + n \cdot k^3 \cdot \log k + \max(n^2, n \cdot k) \cdot G(\max(n^2, n \cdot k)))$$

Bedauerlicherweise läßt sich dieser Term nicht wesentlich vereinfachen, da man im allgemeinen weder $k < n$ noch $n < k$ annehmen kann. Für die SAMPλE-Implementierung ist jedoch $k < 16$ konstant, sodaß sich die Mengenoperationen durch logische Operationen auf einem Maschinenwort realisieren lassen. In diesem Fall gilt

$$T(unify) = O(n^2 \cdot G(n^2)))$$

für den Algorithmus mit occur-check und

$$T(unify) = O(n \cdot G(n))$$

ohne occur-check, was den bekannten Resultaten für ordnungssortierte Unifikation entspricht.

[10] $F(n)$ wächst so rasant, daß $G(n) \leq 5$ für $n \leq 2^{65536}$. Daher spricht man hier gerne von „quasi-linearem“ Aufwand.

Entfernt man die Zeilen 77 bis 79 und überprüft nach dem Aufruf von *unify* den entstanden Graphen auf Zyklen, dann erhält man einen Unifikationsalgorithmus mit der Komplexität $O(n \cdot G(n) + n \cdot k^3 \cdot \log k)$, da sich der Test auf Zyklenfreiheit mit linearem Aufwand in der Anzahl der Knoten und Kanten realisieren läßt. Unterläßt man den Zyklentest, dann erhält man einen Überladungsunifikationsalgorithmus für reguläre Terme, wie er in SAMPλE implementiert ist.

Zum Abschluß beantworten wir noch die Frage, warum die Zeilen 90 bis 93 erlaubt sind, d.h., warum die Operatormengenmarkierungen von *unify* korrekt berechnet werden. Daß die Funktion *cs* die Operatormengenmarkierungen korrekt berechnet, ist offensichtlich, was aber ist mit *unify*? Der Fall, daß ein Term eine Variable ist, wird wg. *cs* korrekt behandelt. Bei der Unifikation zweier Terme mit Konstruktoren wird dem Repräsentanten der vereinigten Äquivalenzklassen die Vereinigung der schon überprüften Operatormengen zugewiesen. Daß dies korrekt ist, zeigt

Proposition 3.36. *Falls θ ein Unifikator von τ_1 und τ_2 ist, dann folgt aus $X_1 \uparrow^o \tau_1$ und $X_2 \uparrow^o \tau_2$ auch $(X_1 \cup X_2) \uparrow^o \theta(\tau_1)$ bzw. $(X_1 \cup X_2) \uparrow^o \theta(\tau_2)$.*

Beweis. Folgt aus den Propositionen 3.5 und 3.18: da $\uparrow^o$ unter Substitutionen abgeschlossen ist, folgt $X_1 \uparrow^o \theta(\tau_1)$, sowie $X_2 \uparrow^o \theta(\tau_1)$ (da $\theta(\tau_1) = \theta(\tau_2)$), und damit auch $(X_1 \cup X_2) \uparrow^o \theta(\tau_1)$. □

3.5 Benutzerdefinierbare Überladungen

Bis hierher haben wir uns nur mit vordefinierten überladenen Operatoren beschäftigt. Es liegt natürlich nahe, ein derart mächtiges Konzept wie induktiv definierte überladene Operatoren auch dem Anwender in die Hand zu geben. Wir werden daher im Folgenden unsere Beispielsprache um die Möglichkeit der Definition überladener Operatoren erweitern und untersuchen, welche Konsequenzen daraus für die Typdeduktion bzw. Typinferenz und die Semantik von Programmen resultieren.

Ausdrücke	M	$::= x \mid \lambda x.M \mid M_1 M_2$
		$\mid$ **let** $x = M_1$ **in** M_2
Deklarationen	D	$::=$ **operator** $x :: \omega$
		$\mid$ **extend** $x :: \sigma$ **with** M
		$\mid$ **let** $x = M$
Programme	P	$::= D; P \mid$ **eval** M

Abbildung 3.11: Syntax von Mini-SAMPλE

3.5.1 Syntax und Typdeduktion

Zusätzlich zur Menge der Ausdrücke, die wir unverändert aus den vorangegangenen Abschnitten übernehmen, führen wir zwei neue syntaktische Klassen ein: Programme und Deklarationen (siehe Abbildung 3.11). Programme bestehen aus einer Semikolon-separierten Liste von Deklarationen, gefolgt von einem Ausdruck, dessen Semantik von den in der Deklarationsliste eingeführten überladenen Operatoren und globalen Bezeichnern abhängt.[11] Diese Sprache nennen wir Mini-SAMPλE, in Anlehnung an Mini-ML (siehe [CDDK86]) und Mini-Haskell (siehe [NS91]).

Wir betrachten zunächst ein einfaches Beispiel, in dem zwei überladene Operatoren $+$ und $*$ deklariert werden und mit Ganzzahl- und Fließkomma-Addition und Multiplikation überladen werden, unter einer initialen Typannahme $\mathit{intadd}, \mathit{intmult} : \mathit{int} \to \mathit{int} \to \mathit{int}, \mathit{realmult}, \mathit{realadd} : \mathit{real} \to \mathit{real} \to \mathit{real}$. Schließlich wird eine Funktion *addsquares* mit zwei Parametern x, y definiert, welche beide Parameter quadriert und die Ergebnisse der Quadratur addiert. Die so definierte Funktion *addsquares*

[11] In Beispielprogrammen schreiben wir die Deklarationen einfach untereinander, ohne das Semikolon explizit als Trennzeichen anzugeben. Dies ist ein Spezialfall der *Einrückungsregel*, die auf P. Landin [Lan66] zurückgeht und z.B. in SAMPλE, Miranda und Haskell verwendet wird: der Gültigkeitsbereich einer Deklaration wird durch das erste relevante Token (Bezeichner, Trennzeichen, Schlüsselwort) der nächsten Zeile beendet, das keine größere Einrückung hat, als das erste Zeichen der die Deklaration beginnenden Zeile.

wird anschließend auf zwei ganzzahlige Werte angewendet.

$$
\begin{array}{l}
\textbf{operator} * :: \$ \to \$ \to \$ \\
\textbf{operator} + :: \$ \to \$ \to \$ \\
\textbf{extend} * :: \mathit{int} \to \mathit{int} \to \mathit{int} \ \textbf{with} \ \mathit{intmult} \\
\textbf{extend} * :: \mathit{real} \to \mathit{real} \to \mathit{real} \ \textbf{with} \ \mathit{realmult} \\
\textbf{extend} + :: \mathit{int} \to \mathit{int} \to \mathit{int} \ \textbf{with} \ \mathit{intadd} \\
\textbf{extend} + :: \mathit{real} \to \mathit{real} \to \mathit{real} \ \textbf{with} \ \mathit{realadd} \\
\textbf{let} \ \mathit{addsquares} = \lambda x.\lambda y.\ x * x + y * y \\
\textbf{eval} \ \mathit{addsquares}\ 3\ 5
\end{array}
$$

Die Deklaration **operator** $x :: \omega$ führt einen neuen überladenen Operator x mit Überladungsschema ω ein, die Deklaration **extend** $x :: \sigma$ **with** M definiert M als Semantik einer neuen Überladungsinstanz für x und **let** $x = M$ dient zur Deklaration globaler Bezeichner. Dabei verlangen wir, daß in jedem Programm die Namen aller globalen Bezeichner einschließlich der überladenen Operatoren paarweise verschieden sind.[12]

Abbildung 3.12 definiert die gültigen Typisierungen für Mini-SAMPλE-Programme mit Hilfe drei verschiedener Typaussagen. $O, \Gamma \vdash^e M : \tau$ definiert die gültigen Typen τ für Ausdrücke M unter der Überladungsannahme O und Typannahme Γ und ist äquivalent zu $\Gamma \vdash^o M : \tau$ (Regeln VAR, ABS, APP und LET). Die Aussage $O, \Gamma \vdash^d D : O', \Gamma'$ beschreibt wohlgeformte Deklarationen (Regeln OP, OV und GLB) und $O, \Gamma \vdash^p P$ definiert typisierbare Programme (Regeln SEQ und VAL).

Programme sind unter einer gegebenen initialen Überladungsannahme typisierbar, falls sich mit Hilfe der Deklarationsregeln aus den Deklarationen eine wohlgeformte, erweiterte Überladungsannahme und Typannahme ableiten lassen und der „auszuwertende“ Ausdruck sich unter diesen Annahmen typisieren läßt.

Man beachte, daß überladene Operatoren unmittelbar nach ihrer Deklaration polymorph verwendet werden dürfen. Daher könnte man die Deklaration der Funktion *addsqares* im o.a. Beispielprogramm auch in schon in der dritten Zeile angeben, ohne

[12] Dies dient der Vereinfachung des Typdeduktionssystems und der Semantikdefinition und kann durch α-Konversion immer erreicht werden. In einer echten Programmiersprache wie z.B. SAMPλE werden zur Typdeduktion ohnehin intern generierte, eindeutige Bezeichner verwendet. Die Identifikation der äquivalenten Bezeichner ist Aufgabe der Bezeichneranalyse (siehe z.B. [HS83]).

[VAR] $$\frac{\tau \prec^o \Gamma(x)}{O, \Gamma \vdash^e x : \tau}$$

[ABS] $$\frac{O, \Gamma \vdash^e M : \tau \to \tau' \qquad O, \Gamma \vdash^e N : \tau}{O, \Gamma \vdash^e MN : \tau'}$$

[APP] $$\frac{O, \Gamma + [x : \tau] \vdash^e M : \tau' \qquad x \notin dom(O)}{O, \Gamma \vdash^e \lambda x.M : \tau \to \tau'}$$

[LET] $$\frac{O, \Gamma \vdash^e M : \tau \qquad O, \Gamma + [x : \mathrm{gen}(\Gamma, \tau)] \vdash^e N : \tau' \qquad x \notin dom(O)}{O, \Gamma \vdash^e \textbf{let } x = M \textbf{ in } N : \tau'}$$

[OP] $$\frac{\sigma = \forall \alpha_{\{x\}}.\{\$ \to \alpha_{\{x\}}\}\omega \qquad x \notin dom(O) \quad x \notin dom(\Gamma)}{O, \Gamma \vdash^d \textbf{operator } x :: \omega : O + [x : \langle \omega, [] \rangle], \Gamma + [x : \sigma]}$$

[OV] $$\frac{\begin{array}{l}\sigma = \mathrm{gen}(\Gamma, \tau) = \forall \overline{\alpha_n}.\{\$ \to f(\overline{\alpha_n})\}\omega \\ O(x) = \langle \omega, d \rangle \wedge d' = d \cup \{f(\overline{\alpha_n})\} \\ O, \Gamma \vdash^e M : \tau \qquad \nexists f(\overline{\beta_n}) \in d\end{array}}{O, \Gamma \vdash^d \textbf{extend } x :: \sigma \textbf{ with } M : O + [x : \langle \omega, d' \rangle], \Gamma}$$

[GLB] $$\frac{O, \Gamma \vdash^e M : \tau \qquad x \notin dom(O) \quad x \notin dom(\Gamma)}{O, \Gamma \vdash^e \textbf{let } x = M : O, \Gamma + [x : \mathrm{gen}(\Gamma, \tau)]}$$

[SEQ] $$\frac{O, \Gamma \vdash^d D : O', \Gamma' \qquad O', \Gamma' \vdash^p P}{O, \Gamma \vdash^p D; P}$$

[VAL] $$\frac{O, \Gamma \vdash^e M : \tau}{O, \Gamma \vdash^p \textbf{eval } M}$$

Abbildung 3.12: Typdeduktionssystem für Mini-SAMPλE

daß sich die Typisierbarkeit bzw. die Semantik des Programms ändern würde. Dies ist eine herausragende Eigenschaft des Typsystems, erlaubt sie doch die *inkrementelle Erweiterbarkeit* der Semantik überladener Operatoren. Dies ermöglicht, ebenso wie der parametrische Polymorphismus, die Definition von Rechenvorschriften, ohne genaue Kenntnis aller möglichen Typen, auf die die Rechenvorschrift zukünftig angewendet werden kann.

An dieser Stelle könnte man nun fragen, warum wir nicht dem gesamten Programm einen Typ zuordnen, d.h. warum $\vdash^p$-Aussagen nicht die Form $O, \Gamma \vdash^p P : \tau$ haben und die beiden Regeln SEQ und VAL nicht wie folgt formuliert wurden:

$$\text{[SEQ']} \quad \frac{O, \Gamma \vdash^d D : O', \Gamma' \qquad O', \Gamma' \vdash^p P : \tau}{O, \Gamma \vdash^p D; P : \tau}$$

$$\text{[VAL']} \quad \frac{O, \Gamma \vdash^e M : \tau}{O, \Gamma \vdash^p \textbf{eval}\ M : \mathrm{gen}(\Gamma, \tau)}$$

Zur Beantwortung dieser Frage betrachten wir das folgende Beispielprogramm P:

$$\begin{array}{l}\textbf{operator}\ x :: \$ \to \$\\ \textbf{extend}\ x :: \mathit{int} \to \mathit{int}\ \textbf{with}\ \lambda y.1\\ \textbf{extend}\ x :: \mathit{real} \to \mathit{real}\ \textbf{with}\ \lambda y.\ 1.0\\ \textbf{eval}\ x\end{array}$$

Für den finalen Ausdruck x dieses Programms können unter der Überladungsannahme $O = [x \mapsto \$ \to \$, \{\mathit{int}, \mathit{real}\}]$ und der Typannahme $\Gamma = [x \mapsto \forall\alpha_{\{x\}}.\alpha_{\{x\}} \to \alpha_{\{x\}}]$ mit der Regel VAL' drei verschiedene Typen deduziert werden: $t_1 = \mathit{int} \to \mathit{int}$, $t_2 = \mathit{real} \to \mathit{real}$ und $t_3 = \forall\alpha_{\{x\}}.\alpha_{\{x\}} \to \alpha_{\{x\}}$, wobei $t_1 \prec^o t_3$ und $t_2 \prec^o t_3$, d.h. t_3 ist ein allgemeinster Typ für den Ausdruck x. Durch 3-malige Anwendung der SEQ'-Regel erhält man dann entsprechend $[], [] \vdash^p P : t_i$ für $i = 1..3$. Nach Definition von $\uparrow^o$ bzw. $\prec^o$ gilt aber nun nicht mehr $t_1, t_2 \prec^{[]} t_3$! Daher müssen wir die Hoffnung auf einen allgemeinsten Typ für Programme bedauerlicherweise begraben, was wiederum zu Aussagen der Form $O, \Gamma \vdash^p P$ in der o.a. Form führt.

Zum Abschluß dieses Abschnitts geben wir noch eine Definition eines überladenen Gleichheitsoperators an, der für alle Typen definiert ist, die mit Hilfe der Typkon-

struktoren *int*, *real*, *list* und $\times$ gebildet werden können:

$$
\begin{array}{l}
\textbf{operator}\ (=) :: \$ \to \$ \to \mathit{bool} \\
\textbf{extend}\ (=) :: \mathit{int} \to \mathit{int} \to \mathit{bool}\ \textbf{with}\ \mathit{inteq} \\
\textbf{extend}\ (=) :: \mathit{real} \to \mathit{real} \to \mathit{bool}\ \textbf{with}\ \mathit{realeq} \\
\textbf{let}\ \mathit{paireq} = \lambda p.\lambda q.\ \mathit{fst}\ p = \mathit{fst}\ q \wedge \mathit{snd}\ p = \mathit{snd}\ q \\
\textbf{extend}\ (=) :: \alpha_{\{=\}} \times \beta_{\{=\}} \to \alpha_{\{=\}} \times \beta_{\{=\}} \to \mathit{bool}\ \textbf{with}\ \mathit{paireq} \\
\textbf{let}\ \mathit{listeq} = Y(\lambda f.\lambda l_1.\lambda l_2.\ \textbf{if}\ \mathit{null}\ l_1\ \textbf{then}\ \mathit{null}\ l_2\ \textbf{else} \\
\qquad\qquad \neg(\mathit{null}\ l_2) \wedge (\mathit{hd}\ l_1 = \mathit{hd}\ l_2) \wedge f(\mathit{tl}\ l_1)\ (\mathit{tl}\ l_2)) \\
\textbf{extend}\ (=) :: \mathit{list}(\alpha_{\{=\}}) \to \mathit{list}(\alpha_{\{=\}}) \to \mathit{bool}\ \textbf{with}\ \mathit{listeq} \\
\textbf{eval}\ \langle\,(1,2)\,\rangle = \langle\,\rangle
\end{array}
$$

3.5.2 Denotationale Semantik

Wir geben nun eine denotationale Semantik für Mini-SAMPλE an und zeigen, daß wohltypisierte Programme nicht zu Laufzeittypfehlen führen. Dazu nutzen wir aus, daß wir in Abschnitt 3.3 ein entsprechendes Resultat für wohltypisierte Ausdrücke bereits bewiesen haben. Wir werden also im wesentlichen eine Semantik für Deklarationen angeben, welche die Voraussetzungen von Satz 3.15 erfüllt.

Wir beginnen zunächst mit einer naheliegenden Semantik, in der Deklarationen von globalen Bezeichnern und Operatoren als Wertumgebungstransformationen interpretiert werden, die bedauerlicherweise aber nicht die Vermeidung von Typfehlern zur Laufzeit garantiert.[13]

Die Deklaration **operator** $x :: \omega$ führt einen neuen Operator x ein, der initial auf keinem Argumenttyp definiert ist. Daher könnte man dieser Deklaration die Bedeutung $[\![\textbf{operator}\ x :: \omega]\!]\eta = \eta\{\lambda\overline{v_n}.v_1 = \bot \to \bot, \mathit{wrong}/x\}$ zuordnen, wobei n die Anzahl der Argumente von x gemäß Überladungsschema ω ist und wir zur Vereinfachung annehmen, daß x im ersten Argument überladen ist. D.h., **operator** $x :: \omega$ erweitert die Wertumgebung η um die Funktion, die $\bot$ in $\bot$ abbildet. Die Deklaration **extend** $x :: \sigma$ **with** M könnte man dann als Erweiterung der

[13] Dieser Ansatz wurde für das in [OWW95] definierte Überladungstypsystem verwendet, obwohl wir schon in der ersten Veröffentlichung zur Überladungstypisierung ([Kae88]) auf das Problem hingewiesen haben.

dem Operator in η zugeordneten Funktion für den Instanzargumenttyp f interpretieren: $[\![\textbf{extend } x :: \sigma \textbf{ with } M]\!]\eta = \eta\{g/x\}$, wobei $g = \lambda\overline{v_n}.(v_1 \in \boldsymbol{D}_f \to [\![M]\!]\eta,\ \eta(x))\overline{v_n}$. Die Semantik von **let** $x = M$ wäre dann analog zum Let-Konstrukt in Ausdrücken definiert: $[\![\textbf{let } x = M]\!]\eta = \eta\{[\![M]\!]\eta/x\}$.

Um zu sehen, warum diese Semantik die Typfehlerfreiheit nicht garantieren kann, betrachten wir eine vereinfachte Definition des Gleichheitsoperators:

operator $(=) :: \$ \to \$ \to \mathit{bool}$
extend $(=) :: \mathit{int} \to \mathit{int} \to \mathit{bool}$ **with** *inteq*
let $\mathit{paireq} = \lambda p.\lambda q.\ \mathit{fst}\ p = \mathit{fst}\ q \wedge \mathit{snd}\ p = \mathit{snd}\ q$
extend $(=) :: \alpha_{\{=\}} \times \beta_{\{=\}} \to \alpha_{\{=\}} \times \beta_{\{=\}} \to \mathit{bool}$ **with** *paireq*

Durch Anwenden der o.a. Definitionsgleichungen erhält man die folgende Sequenz von Wertumgebungen:

$$\eta_1 = \eta_o + [\,(=) \mapsto \lambda v_1.\lambda v_2.v_1 = \bot \to \bot, \mathit{wrong}\,]$$
$$\eta_2 = \eta_1 + [\,(=) \mapsto \lambda v_1.\lambda v_2.v_1 \in \boldsymbol{D}_{\mathit{int}} \to [\![\mathit{inteq}]\!]\eta_1\, v_1 v_2,\ \eta_1(=)v_1 v_2\,]$$
$$\eta_3 = \eta_2 + [\,\mathit{paireq} \mapsto \ldots \eta_2(=) \ldots \eta_2(=) \ldots\,]$$
$$\eta_4 = \eta_3 + [\,(=) \mapsto \lambda v_1.\lambda v_2.v_1 \in \boldsymbol{D}_\times \to [\![\mathit{paireq}]\!]\eta_3\, v_1 v_2),\ \eta_3(=)v_1 v_2\,]$$

Man sieht, daß die Bedeutung des Gleichheitsoperators im Rumpf einer Erweiterung eingeschränkt wird auf die Bedeutung, die für den Operator bis zu diesem Zeitpunkt in der Wertumgebung bestimmt wurde. Damit kann man den hier definierten Operator zwar auf ganze Zahlen und auf Paare von ganzen Zahlen anwenden, die gemäß Typsystem erlaubte Anwendung auf Paare von Paaren ganzer Zahlen $(\mathit{int} \times \mathit{int}) \times (\mathit{int} \times \mathit{int})$ führt jedoch zu einem Laufzeittypfehler!

Die intendierte Semantik des Gleichheitsoperators ist die Lösung der rekursiven Gleichung

$$\eta(=) = \lambda v_1.\lambda v_2.v_1 \in \boldsymbol{D}_{\mathit{int}} \to [\![\mathit{inteq}]\!]\eta\, v_1 v_2,$$
$$v_1 \in \boldsymbol{D}_\times \to [\![\mathit{paireq}]\!]\eta\, v_1 v_2,$$
$$\mathit{wrong}$$

Die Bedeutung eines global definierten Bezeichners kann also erst bestimmt werden, wenn alle Überladungsinstanzen bekannt sind. Da die Instanzdefinitionen zudem eine gegenseitige Abhängigkeit zwischen der Semantik unterschiedlicher Operatoren

erzeugen können, und in der Regel auch die Semantik der übrigen global deklarierten Bezeichner von der Semantik der überladenen Operatoren abhängt, muß der Fixpunkt einer rekursiven Gleichung für die Wertumgebung zur Auswertung des Programmausdrucks bestimmt werden.

Wir definieren daher die Semantik von Deklarationen als Transformation eines Paares, bestehend aus einer Wertumgebung $e \in \mathbf{Id} \to \mathit{Env}$ und einer *Deklarationsumgebung* $d \in \mathit{DclEnv}$, wobei der semantische Bereich *DclEnv* aus endlichen Abbildungen besteht, die Bezeichnern entweder eine Funktion von Wertumgebungen auf denotierbare Werte zuordnet (für global let-definierte Bezeichner), oder ein Paar aus einem Überladungsschema und einer endlichen Abbildung von Typkonstruktoren auf Funktionen von Wertumgebungen auf denotierbare Werte.

$$\mathit{DclEnv} = \mathbf{Id} \mapsto ((\mathit{Env} \to \mathbf{DV}) \oplus (T_F(\mathcal{V}^{\$}) \times F \mapsto \mathit{Env} \to \mathbf{DV}))$$

Die Semantikfunktion $\mathcal{D}$, welche die Bedeutung von Deklarationen beschreibt, ist wie folgt definiert:

$$\begin{array}{lll}
\mathcal{D}[\![D]\!] : \mathit{Env} \times \mathit{DclEnv} \to \mathit{Env} \times \mathit{DclEnv} & & \\
\mathcal{D}[\![\textbf{operator } x :: \omega]\!](e,d) & = & (e, d + [\, x \mapsto \langle\, \omega, [\,]\,\rangle\,]) \\
\mathcal{D}[\![\textbf{extend } x :: \sigma \textbf{ with } M]\!](e,d) & = & (e, d') \\
 & & \text{wobei} \\
 & & \quad \sigma = \forall \overline{\alpha_n}.\{\$ \to f(\overline{\alpha_n})\}\omega \\
 & & \quad \langle \omega, s \rangle = d(x) \\
 & & \quad d' = d + [\, x \mapsto \langle\, \omega, s \cup [\, f \mapsto \mathcal{E}[\![M]\!]\,]\,\rangle\,] \\
\mathcal{D}[\![\textbf{let } x = M]\!](e,d) & = & (e, d + [\, x \to \mathcal{E}[\![M]\!]\,])
\end{array}$$

Das Operator-Konstrukt ordnet dem Bezeichner x in der Deklarationsumgebung ein Paar aus Überladungsschema und einer leeren Abbildung zu (da x noch für keinen Typ definiert ist), das Extend-Konstrukt erweitert diese Abbildung um die Semantik der Überladungsinstanz, ohne die Wertumgebung zu binden. Das Let-Konstrukt erweitert die Deklarationsumgebung um die Zuordnung $x \mapsto \mathcal{E}[\![M]\!]$. Auch hier wird die Bindung der Wertumgebung verzögert.

Ohne Überraschungen wird die Semantik von Programmen definiert:

$$\begin{aligned}
&\mathcal{P}[\![P]\!] : \mathit{Env} \times \mathit{DclEnv} \to \mathbf{DV} \\
&\mathcal{P}[\![D; P]\!]\,(e,d) \quad = \mathcal{P}[\![P]\!]\,(\mathcal{D}[\![D]\!]\,(e,d)) \\
&\mathcal{P}[\![\mathbf{eval}\ M]\!]\,(e,d) = \mathcal{E}[\![M]\!]\,(\mathit{closure}(e,d))
\end{aligned}$$

Die Funktion *closure* ist das Kernstück der Programmsemantik. Sie erzeugt aus einer Wertumgebung e, welche die Bedeutung vordefinierter Funktionen enthält und der durch die Funktion $\mathcal{D}$ erzeugten Deklarationsumgebung d, die Wertumgebung η, mit der die Bedeutung des Programmausdruck **eval** M bestimmt wird. Sie wird definiert durch

$$\begin{aligned}
&\mathit{closure} : \mathit{Env} \times \mathit{DclEnv} \to \mathit{Env} \\
&\mathit{closure}(e,d) = \underline{\mathit{fix}}\ H \\
&\quad \text{wobei} \\
&\qquad H\,\eta = e \cup [\, o_1 \mapsto f_1\eta, \ldots, o_n \mapsto f_n\eta \,] \cup [\, x_1 \mapsto g_1\eta, \ldots, x_m \mapsto g_m\eta \,] \\
&\qquad \{o_1, \ldots, o_n\} = \{x \in \mathit{dom}(d) \mid d(x) \in F \mapsto \mathit{Env} \to \mathbf{DV}\} \\
&\qquad f_1, \ldots, f_n = \mathit{resolve}(d(o_1)), \ldots, \mathit{resolve}(d(o_n)) \\
&\qquad \{x_1, \ldots, x_m\} = \{x \in \mathit{dom}(d) \mid d(x) \in \mathit{Env} \to \mathbf{DV})\} \\
&\qquad g_1, \ldots, g_m = d(x_1), \ldots, d(x_m)
\end{aligned}$$

und die Definition der Funktion *resolve* lautet:

$$\begin{aligned}
&\mathit{resolve} : T_F(\mathcal{V}^{\$}) \times (F \mapsto \mathit{Env} \to \mathbf{DV}) \to \mathit{Env} \to \mathbf{DV} \\
&\mathit{resolve}(\omega, d)\,\eta = \\
&\qquad \lambda\overline{v_n}.\begin{cases}
\bot & \text{falls } v_i = \bot \text{ für ein } i \in J \\
f_1\,\eta\,\overline{v_n} & \text{falls } \forall i \in J : v_i \trianglelefteq \boldsymbol{D}_{c_1} \\
\vdots & \\
f_m\,\eta\,\overline{v_n} & \text{falls } \forall i \in J : v_i \trianglelefteq \boldsymbol{D}_{c_m} \\
\mathit{wrong} & \text{sonst}
\end{cases} \\
&\qquad \text{wobei} \\
&\qquad\quad \omega = \omega_1 \to \ldots \to \omega_n \to \omega_r \\
&\qquad\quad J = \{i \in 1..n \mid \omega_i = \$\} \\
&\qquad\quad \overline{c_m} = \mathit{dom}(d) \\
&\qquad\quad \overline{f_m} = \mathit{rng}(d)
\end{aligned}$$

Damit gelangen wir zu

Satz 3.37 (Wohltypisierte Mini-SAMPλE-Programme enthalten keine Laufzeittypfehler). *Sei* $\Gamma_0, [\,] \vdash^p P$ *eine gültige Typisierung mit* $\mathcal{FV}(\Gamma_0) = \emptyset$, $\mathcal{FV}(P) \subseteq dom(\Gamma_o)$ *und* $\forall x \in dom(\eta_0).\ \eta_0(x) \in \mathcal{T}[\![\Gamma_0(x)]\!]\emptyset$, *dann folgt* $\mathcal{P}[\![P]\!](\eta, [\,]) \in \mathbf{I}(\mathbf{DV})$.

Für den Beweis benötigen wir die folgenden Eigenschaften der Überladungstypisierung von Mini-SAMPλE-Programmen.

Definition 3.38. Seien O und O' Überladungsannahmen, Γ und Γ' Typannahmen die kompatibel zu O bzw. O' sind. (Γ', O') ist eine *Erweiterung* von (Γ, O), falls $\Gamma \subseteq \Gamma'$ und aus $O(x) = \langle\, \omega, s \,\rangle$ folgt $O'(x) = \langle\, \omega, s' \,\rangle$ und $s \subseteq s'$.

Proposition 3.39. *Falls* Γ', O' *eine Erweiterung von* Γ, O *ist, dann gilt:*

$$\tau_1 \leq^o \tau_2 \Rightarrow \tau_1 \leq^{o'} \tau_2 \tag{1}$$

$$\sigma_1 \prec^o \sigma_2 \Rightarrow \sigma_1 \prec^{o'} \sigma_2 \tag{2}$$

$$\Gamma, O \vdash^e M : \tau \Rightarrow \Gamma', O' \vdash^e M : \tau \tag{3}$$

$$\Gamma, O \vdash^d \Gamma', O' \Rightarrow \Gamma', O' \text{ ist eine Erweiterung von } \Gamma, O \tag{4}$$

Beweis. (1) und (2) folgen direkt aus der Definition von $\uparrow^o$, (3) aus (2) durch Induktion über M und (4) durch genaue Examination der Regeln OP, OV und GLB, wobei insbesondere auch die Prämissen $x \notin dom(O)$ bzw. $x \notin dom(\Gamma)$ angewendet werden müssen. □

Beweis. (von Satz 3.37)
Jedes Mini-SAMPλE-Programm ist von der Form $D_1; \ldots; D_l; \mathbf{eval}\ M$. Dessen Semantik wird durch $\mathcal{E}[\![M]\!]\eta_c$ definiert, wobei $\eta_c = closure(e_l, d_l)$ und für alle $i = 1..l$ gilt $(e_i, d_i) = \mathcal{D}[\![D_i]\!](e_{i-1}, d_{i-1})$ mit $d_0 = [\,]$ und $e_0 = \eta_0$. Weiterhin existieren für $i = 1..l$ Überladungs- und Typannahmen O_i, Γ_i, sodaß $O_{i-1}, \Gamma_{i-1} \vdash^d O_i, \Gamma_i$ und $O_l, \Gamma_l \vdash^e M : \tau$.

Wir zeigen durch Fixpunktinduktion, daß η_c eine zu Γ kompatible Wertumgebung ist, d.h., $\eta_c(x) \in \mathcal{T}_o^{\perp}[\![\Gamma(x)]\!]\emptyset$ für alle $x \in dom(\eta_c)$. Daraus folgt aufgrund von Satz 3.15 sofort $\mathcal{E}[\![M]\!]\eta_c \in \mathcal{T}_o^{\perp}[\![\mathrm{gen}(\Gamma_l, \tau)]\!]$ und also ist $\mathcal{E}[\![M]\!]\eta_c$ ein Ideal und enthält damit keine Typfehler.

Zunächst beachte man, daß das kleinste Element $\bot_{Env}$ in $Env = \mathbf{Id} \mapsto \mathbf{DV}$ gegeben ist durch die Funktion $\lambda x.\bot_{\mathbf{DV}}$. Nun gilt aber $\bot_{\mathbf{DV}} \in \mathcal{T}_o^{\bot}[\![\sigma]\!]\emptyset$ für jedes geschlossene Typschema σ. Da Γ_l eine geschlossene Typannahme ist, ist $\bot_{Env}$ eine zu Γ_l kompatible Wertumgebung.

Für den Induktionschritt nehmen wir an, daß η eine kompatible Wertumgebung ist und zeigen, daß $H(\eta)$ gleichfalls kompatibel ist. $H(\eta)$ zerfällt in drei disjunkte Teile: die ursprüngliche Wertumgebung η_0, eine Teilumgebung welche die Let-deklarierten Bezeichner $x_1, \ldots, x_m$ enthält, und die Zuordnungen für die überladenen Bezeichner $o_1, \ldots, o_n$.[14]

Daß η_0 kompatibel zu Γ_l ist, folgt aus Teil 3 und 4 der Proposition 3.39. Für einen Let-deklarierten Bezeichner x_i, dem in $H(\eta)$ die Funktion $g_i = d(x_i)$ zugeordnet wird, wissen wir aufgrund der Definition von $\mathcal{D}$ und der Typisierbarkeit des Gesamtprogramms, daß ein Ausdruck M_i und ein Typausdruck τ_i existiert, sodaß $g_i = \mathcal{E}[\![M_i]\!]$ und $\Gamma_{i+1}(x_i) = \mathrm{gen}(\Gamma_i, \tau_i)$. Aufgrund von Proposition 3.39 und der Geschlossenheit von Γ_l folgt aber auch $\Gamma_l(x_i) = \mathrm{gen}(\Gamma_l, \tau_i)$. Da η nach Voraussetzung kompatibel ist zu Γ_l, gilt aber $\mathcal{E}[\![M_i]\!]\eta \in \mathcal{T}_o^{\bot}[\![\mathrm{gen}(\Gamma_l, \tau_i)]\!]\emptyset$ und damit auch $H(\eta)(x_i) \in \mathcal{T}_o^{\bot}[\![\mathrm{gen}(\Gamma_l, \tau_i)]\!]\emptyset$.

Sei o_i ein überladener Bezeichner. Dann gilt nach Definition des Deduktionssystems $O_l(o_i) = \langle \omega, s \rangle$ und $\Gamma_l(o_i) = \sigma_{o_i}$ wobei $\sigma_{o_i} = \forall \alpha_{\{o_i\}}.\{\$ \mapsto \alpha_{\{o_i\}}\}\omega$ und wir müssen zeigen, daß für $f_{o_i} = \mathit{resolve}(\omega, d)$ auch $f_{o_i}\eta \in \mathcal{T}_o^{\bot}[\![\sigma_{o_i}]\!]\emptyset$ gilt. Aufgrund der Definition der Typisierbarkeit existieren Instanzdeklarationen **extend** $o_i :: \sigma^{o_i}$ **with** $M_j^{o_i}$ für $j = 1..k$, wobei $k = |s|$. Nach Definition von $\mathcal{D}$ gilt $d(o_i) = \langle \omega, s' \rangle$ mit $s' = [\, c_1 \mapsto \mathcal{E}[\![M_1^{o_i}]\!], \ldots, c_k \mapsto \mathcal{E}[\![M_k^{o_i}]\!] \,] \rangle$ und, da $O_j, \Gamma_j \vdash^e M^{o_i} : \tau_j$ mit $\sigma_j^{o_i} = \mathrm{gen}(O_j, \Gamma_j)$ aufgrund von Proposition 3.39 auch $O_l, \Gamma_l \vdash^e M_j^{o_i} : \tau_j$ mit $\sigma_j^{o_i} = \mathrm{gen}(O_l, \Gamma_l)$ impliziert, gilt wg. der Geschlossenheit von $\sigma_j^{o_i}$ und der Induktionshypothese auch $\mathcal{E}[\![M_j^{o_i}]\!]\eta \in \mathcal{T}_o^{\bot}[\![\sigma_j^{o_i}]\!]\emptyset$.

Nach Definition von $\mathcal{T}_o^{\bot}$ gilt $f_{o_i}\eta \in \mathcal{T}_o^{\bot}[\![\sigma_{o_i}]\!]\emptyset$, falls einer der beiden folgenden Fälle

[14] Hier kommt wiederum die paarweise Verschiedenheit global deklarierter Bezeichner zum Tragen.

zutrifft:

$$\mathcal{I}_O(o_i) = \emptyset \text{ und } f_{o_i}\eta \in \mathcal{T}_o^{\perp}[\![\omega]\!]\emptyset\{\{\perp_{\mathbf{DV}}\}/\alpha_{\{o_i\}}\} \quad \text{oder} \tag{1}$$

$$\mathcal{I}_O(o_i) \neq \emptyset \text{ und } f_{o_i}\eta \in \bigcap_{t \in T_F(\emptyset) \wedge o_i \uparrow^o t} \mathcal{T}_o^{\perp}[\![\omega]\!]\emptyset\{\mathcal{T}_o^{\perp}[\![t]\!]\emptyset/\alpha_{\{o_i\}}\} \tag{2}$$

Fall 1 stellt die Anforderung, daß $f_{o_i}\eta\overline{v_n} \in \mathcal{T}_o^{\perp}[\![\omega_r]\!]\emptyset\{\{\perp_{\mathbf{DV}}\}/\alpha_{\{o_i\}}\}$, falls einer der an überladenen Argumentpositionen übergebenen Werte $\perp_{\mathbf{DV}}$ ist. Nach Definition von *resolve* ist jedoch $f_{o_i}\eta\overline{v_n} = \perp_{\mathbf{DV}}$, falls eines der Argumente $\perp_{\mathbf{DV}}$ ist. Da $\perp_{\mathbf{DV}}$ in jedem Ideal liegt, gilt auch $f_{o_i}\eta\overline{v_n} \in \mathcal{T}_o^{\perp}[\![\omega_r]\!]\emptyset\{\{\perp_{\mathbf{DV}}\}/\alpha_{\{o_i\}}\}$.

Für Fall 2 genügt es zu zeigen, daß $f_{o_i}\eta \in \mathcal{T}_o^{\perp}[\![\{\alpha_{\{o_i\}} \mapsto t\}\sigma^{o_i}]\!]\emptyset$ für alle $t \in T_F(\emptyset)$, die als Argumenttypen für o_i zulässig sind, d.h, die $o_i \uparrow^o t$ erfüllen. Nach Definition von $\uparrow^o$ muß t von der Form $c_j(\overline{t_n})$ sein, für ein $c_j(\overline{\alpha_n}) \in s$. Dann ist nach Definition von $\mathcal{D}$ aber auch $c_j \in s'$. Aufgrund der Voraussetzung, daß $\mathcal{T}_o^{\perp}[\![c_j(\overline{t_n})]\!]\emptyset \subseteq \boldsymbol{D}_{c_j}$, gilt nach Definition von *resolve* aber $f_{o_i}\eta = \lambda\overline{v_n}.\mathcal{E}[\![M_j^{o_i}]\!]\eta\,\overline{v_n}$, was äquivalent ist zu $f_{o_i}\eta = \mathcal{E}[\![M_j^{o_i}]\!]\eta$. Es gilt $f_{o_i}\eta \in \mathcal{T}_o^{\perp}[\![\{\overline{\alpha_n} \mapsto \overline{t_n}\}\sigma_j^{o_i}]\!]\emptyset$ und da $\{\overline{\alpha_n} \mapsto \overline{t_n}\}\sigma_j^{o_i} = \{\alpha_{\{o_i\}} \mapsto t\}\sigma^{o_i}$, gilt auch $f_{o_i}\eta \in \mathcal{T}_o^{\perp}[\![\{\alpha_{\{o_i\}} \mapsto t\}\sigma^{o_i}]\!]\emptyset$, wie verlangt. □

3.5.3 Dynamische Überladungsauflösung

Die im letzten Abschnitt definierte Semantik hat den Vorteil, die Semantik von Ausdrücken gegenüber dem Fall der vordefinierten überladenen Operatoren nicht anpassen zu müssen und damit die Wohldefiniertheit von Typisierungen direkt folgern zu können. Jedoch ist die Semantik nicht inkrementell, d.h. die Bedeutung der Ausdrücke auf der rechten Seite von globalen Definitionen hängt von dem gesamten Programm ab. Daher ist auch die Herleitung einer konkreten Implementierung nicht trivial. Diesen Nachteil kann man vermeiden, wenn man die Semantikfunktion für Ausdrücke um einen zusätzlichen Parameter erweitert, der die Bedeutung von überladenen Bezeichnern durch eine endliche Abbildung von Typkonstruktoren auf denotationale Werte darstellt. Der denotationale Wert eines Bezeichners hängt dann im Fall einer überladenen Funktion von diesem zusätzlichen Parameter ab.

Wir nennen diesen zusätzlichen Parameter *Operatorumgebung*, den zugeordneten semantischen Bereich bezeichnen wir mit *OpEnv*. Er erfüllt die Gleichung

$$OpEnv = \mathbf{Id} \mapsto F \mapsto OpEnv \to \mathbf{DV}$$

Intuitiv betrachtet, bildet $\vartheta \in \mathit{OpEnv}$ überladene Bezeichner auf Funktionen ab, die bei Anwendung auf einen Typkonstruktor eine Funktion liefern, die eine Operatorumgebung auf eine „echte“ Funktion aus $\mathbf{DV} \to \mathbf{DV}$ abbilden. Dabei sind die zulässigen Typkonstruktoren gerade jene, die dem überladenen Bezeichner in der Überladungsannahme zugeordnet wurden.

Global definierte Bezeichner werden als Elemente des Bereiches $\mathit{OpEnv} \to \mathbf{DV}$ behandelt, was eine Änderung der Definition von Wertumgebungen nach sich zieht:

$$\mathit{Env}' = \mathbf{Id} \mapsto (\mathbf{DV} \oplus (\mathit{OpEnv} \to \mathbf{DV}))$$

Damit definieren wir die Semantik von Ausdrücken nun wie folgt:

$$\begin{array}{ll}
\mathcal{E}'[\![M]\!] : \mathit{Env}' \to \mathit{OpEnv} \to \mathbf{DV} & \\
\mathcal{E}'[\![x]\!]\,\eta\,\vartheta & = (\eta(x) \subseteq \mathit{OpEnv} \to \mathbf{DV}) \to \eta(x)\vartheta,\ \eta(x) \\
\mathcal{E}'[\![\lambda x.M]\!]\,\eta\,\vartheta & = (\lambda v.\mathcal{E}'[\![M]\!]\,\eta\{v/x\}\,\vartheta) \\
\mathcal{E}'[\![M_1 M_2]\!]\,\eta\,\vartheta & = f \subseteq \boldsymbol{D}_{\to} \to (f|\boldsymbol{D}_{\to})\,v,\ \mathit{wrong} \\
 & \quad \text{wobei } f = \mathcal{E}'[\![M_1]\!]\,\eta\,\vartheta,\ v = \mathcal{E}'[\![M_2]\!]\,\eta\,\vartheta \\
\mathcal{E}'[\![\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2]\!]\,\eta\,\vartheta & = \mathcal{E}'[\![M_2]\!]\,\eta\{\mathcal{E}'[\![M_1]\!]\,\eta\,\vartheta/x\}\,\vartheta
\end{array}$$

Man beachte, daß die Operatorumgebung ϑ für die Bestimmung des Wertes eines Ausdrucks unverändert an die Unterausdrücke weitergereicht wird und sozusagen ein globaler Parameter einer entsprechenden Interpretationsfunktion wäre.[15] Die Werte Lambda- und Let-gebundener Bezeichner liegen immer in $\mathbf{DV}$ und werden wie im Falle nicht überladener Ausdrücke in der Wertumgebung fixiert.

Deklarationen erweitern Paare aus Wert- und Operatorumgebungen, wobei je nach Art der Deklaration Wert- und/oder Operatorumgebung verändert wird.

$$\begin{array}{ll}
\mathcal{D}'[\![D]\!] : \mathit{Env}' \times \mathit{OpEnv} \to \mathit{Env}' \times \mathit{OpEnv} & \\
\mathcal{D}'[\![\mathbf{operator}\ x :: \omega]\!]\,(\eta,\vartheta) & = (\eta\{\mathit{resolve}'(\omega,x)/x\},\ \vartheta\{[\,]/x\}) \\
\mathcal{D}'[\![\mathbf{extend}\ x^{\omega} :: \sigma\ \mathbf{with}\ M]\!]\,(\eta,\vartheta) & = (\,\eta,\ \vartheta\{(\vartheta x)\{\mathcal{E}'[\![M]\!]\,\eta/f\}/x\}\,) \\
 & \quad \text{wobei } \eta(x) \in \mathit{OpEnv} \to \mathbf{DV} \text{ und} \\
 & \quad \text{falls } \sigma = \forall\overline{\alpha_n}.\{\$ \to f(\overline{\alpha_n})\}\omega \\
\mathcal{D}'[\![\mathbf{let}\ x = M]\!]\,(\eta,\vartheta) & = (\eta\{\mathcal{E}'[\![M]\!]\,\eta/x\},\vartheta)
\end{array}$$

[15] Siehe dazu auch z.B. [Sch85]

Die Deklaration eines neuen überladenen Operators x ordnet dem Bezeichner in der Wertumgebung eine Abbildung $f = \mathit{resolve}'(\omega, x)$ zu, die eine Operatorumgebung akzeptiert, die die Bedeutung sämtlicher an der Verwendungsstelle definierten Überladungsinstanzen enthält und eine Funktion liefert, die einen Typkonstruktor akzeptiert und auf eine Funktion in **DV** abbildet, während dem Bezeichner x in der Operatorumgebung die leere Abbildung zugeordnet wird. Die Funktion $\mathit{resolve}'$ ist wie folgt definiert:

$$
\begin{array}{l}
\mathit{resolve}' : T_F(\mathcal{V}^{\$}) \times Id \to \mathit{OpEnv} \to \mathbf{DV} \\
\mathit{resolve}'(\omega_1 \to \ldots \to \omega_n \to \omega_r, x)\ \vartheta = \\
\qquad \lambda\overline{v_n}.\begin{cases}
\bot & \text{falls } v_i = \bot \text{ für ein } i \in J \\
((\vartheta\ x)\ f\ \vartheta)\ \overline{v_n} & \text{falls } x \in \mathit{dom}(\vartheta) \text{ und} \\
 & \quad \exists f \in \mathit{dom}(\vartheta(x)).\ \forall i \in J.\ v_i \underline{\in} \boldsymbol{D}_f \\
\mathit{wrong} & \text{sonst}
\end{cases} \\
\text{wobei } J = \{i \in 1..n \mid \omega_i = \$\}
\end{array}
$$

Die Deklaration einer neuen Überladungsinstanz für x erweitert die dem Bezeichner x in der Operatorumgebung zugeordnete Abbildung um das Paar $(f, \mathcal{E}'[\![M]\!]\eta)$, die Deklaration eines globalen Bezeichners x ordnet x in der Wertumgebung den Wert $\mathcal{E}'[\![M]\!]\eta$ zu.

Die Semantik von Programmen ist ähnlich wie im letzten Abschnitt definiert.

$$
\begin{array}{l}
\mathcal{P}'[\![P]\!] : \mathit{Env}' \times \mathit{OpEnv} \to \mathbf{DV} \\
\mathcal{P}'[\![D; P]\!]\ (\eta, \vartheta) \quad = \mathcal{P}'[\![P]\!]\ (\mathcal{D}'[\![D]\!]\ (\eta, \vartheta)) \\
\mathcal{P}'[\![\textbf{eval}\ M]\!]\ (\eta, \vartheta) = \mathcal{E}'[\![M]\!]\ \eta\ \vartheta
\end{array}
$$

Damit gelangen wir zu

Vermutung 3.40 (Die beiden definierten Semantiken für Mini-SAMPλE sind äquivalent). $\mathcal{FV}(\Gamma_0) = \emptyset \wedge \mathcal{FV}(P) \subseteq \Gamma_0 \wedge \forall x \in \mathit{dom}(\eta_0).\, \eta_0(x) \in \mathcal{T}[\![\Gamma_0(x)]\!]\emptyset \wedge \Gamma_0, [\,] \vdash^p P \Rightarrow \mathcal{P}'[\![P]\!](\eta_0, [\,]) = \mathcal{P}[\![P]\!](\eta_0, [\,])$.

Definition 3.41. Sei $\eta \in \mathit{Env}$ und $(\eta', \vartheta) \in \mathit{Env}' \times \mathit{OpEnv}$. Die Wertumgebung η *approximiert* (η', ϑ) (geschrieben $\eta \sqsubseteq (\eta', \vartheta)$), falls für alle $x \in \mathit{dom}(\eta)$ eine der

beiden folgenden Bedingungen erfüllt ist:

$$\eta'(x) \in \mathit{OpEnv} \to \mathbf{DV} \wedge \eta(x) \sqsubseteq \eta'(x)\vartheta \tag{ap1}$$

$$\eta'(x) \in \mathbf{DV} \wedge \eta(x) \sqsubseteq \eta'(x) \tag{ap2}$$

Analog definieren wir die Äquivalenz $\eta \equiv (\eta', \vartheta)$, indem wir $\sqsubseteq$ durch $=$ ersetzen.

Lemma 3.42. *Die Semantikdefinitionen von Ausdrücken sind kompatibel falls die Wertumgebungen und Operatorumgebung kompatibel sind:*

$$(1) \quad \eta \sqsubseteq (\eta', \vartheta) \Rightarrow \mathcal{E}[\![M]\!]\eta \sqsubseteq \mathcal{E}'[\![M]\!]\eta'\vartheta$$

$$(2) \quad \eta \equiv (\eta', \vartheta) \Rightarrow \mathcal{E}[\![M]\!]\eta = \mathcal{E}'[\![M]\!]\eta'\vartheta$$

Beweis. (Durch Induktion über M) Wir betrachten nur (1), der Fall (2) folgt analog. Für $M \equiv x$ folgt die Behauptung direkt aus $\eta \sqsubseteq (\eta', \vartheta)$. Für $M \equiv M_1\, M_2$ folgt aus der Definition von $\mathcal{E}, \mathcal{E}'$ und der Induktionshypothese $\mathcal{E}[\![M_1]\!]\eta = f$, $\mathcal{E}'[\![M_1]\!]\eta\vartheta = g$, $\mathcal{E}[\![M_2]\!]\eta = v$, $\mathcal{E}'[\![M_2]\!]\eta'\vartheta = w$, mit $f \sqsubseteq g$ und $v \sqsubseteq w$. Aufgrund der Definition von $\sqsubseteq$ für Funktionen gilt $f(v) \sqsubseteq g(v)$ und aufgrund der Monotonie von g auch $g(v) \sqsubseteq g(w)$ und damit $\mathcal{E}[\![M_1 M_2]\!]\eta \sqsubseteq \mathcal{E}'[\![M_1 M_2]\!]\eta'\vartheta$. Für $M \equiv \lambda x.N$ gilt: $\mathcal{E}[\![\lambda x.N]\!] = f$, wobei $f(v) = \mathcal{E}[\![N]\!]\eta\{v/x\}$ und $\mathcal{E}'[\![\lambda x.M]\!] = g$, wobei $g(v) = \mathcal{E}[\![N]\!]\eta'\{v/x\}\vartheta$. Nun gilt $f \sqsubseteq g \iff f(v) \sqsubseteq g(v)$ für alle $v \in \mathbf{DV}$. Da $\eta\{v/x\} \sqsubseteq (\eta'\{v/x\}, \vartheta)$ können wir die Induktionshypothese anwenden und erhalten $f(v) \sqsubseteq (v)$. Der Fall $M \equiv \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ folgt analog. □

Satz 3.43 („Äquivalenz" der Semantikdefinitionen).

$$\eta_0 \sqsubseteq (\eta_0', \vartheta_0) \Rightarrow \mathcal{P}[\![P]\!](\eta_0, [\,]) \sqsubseteq \mathcal{P}'[\![P]\!](\eta_0', \vartheta_0).$$

Beweis. Wir zeigen durch Berechnungsinduktion, daß $\eta_l = \mathit{closure}(e_l, d_l)$ das Paar (η', ϑ) approximiert. Aufgrund von Lemma 3.42 folgt dann $\mathcal{E}[\![M]\!]\eta \sqsubseteq \mathcal{E}'[\![M]\!]\eta_l'\vartheta_l$ und damit $\mathcal{P}[\![P]\!](\eta_0, [\,]) \sqsubseteq \mathcal{P}'[\![P]\!](\eta_0, \vartheta_0)$.

Offensichtlich gilt $\bot_{\mathit{Env}} \sqsubseteq (\eta_l, \vartheta_l)$. Sei also $\eta \sqsubseteq (\eta_l', \vartheta_l)$. Wir zeigen $H(\eta) \sqsubseteq (\eta_l', \vartheta_l)$. Nach Konstruktion zerfällt $H(\eta)$ in Zuordnungen für Bezeichner, die schon in η_0 vorhanden sind, let-eingeführte Bezeichner und überladene Operatoren. Aufgrund von $H(\eta)_{|dom(\eta_0)} = \eta_0$, gilt nach Voraussetzung $H(\eta)_{|dom(\eta_0)} \sqsubseteq (\eta_0', \vartheta_0)$. Für let-eingeführte Bezeichner x ist $H(\eta)(x) = \mathcal{E}[\![M_x]\!]\eta$, wobei M_x die rechte Seite der

let-Definition ist. Aufgrund der Definition von $\mathcal{D}'$ ist $x \in dom(\eta'_l)$ und es gilt $\eta'_l(x) = \mathcal{E}'[\![M_x]\!]\eta_j$ für ein $\eta_j \subset \eta'_l$. Damit gilt aber auch $\mathcal{E}'[\![M_x]\!]\eta_j = \mathcal{E}'[\![M_x]\!]\eta_l$. Aufgrund der Induktionshypothese $\eta \sqsubseteq (\eta'_l, \vartheta_l)$ folgt aus Lemma 3.42 aber $\mathcal{E}[\![M_x]\!]\eta_l = \mathcal{E}'[\![M_x]\!]\eta_l\vartheta_l$ (d.h. x erfüllt (ap1)).

Sei x ein Operator. Dann ist $\eta'_l(x) = resolve(\omega, x)$ und $\vartheta_l(x) = [c \mapsto \mathcal{E}'[\![M_c]\!]\eta'_c | x \in m(c)]$, wobei M_c der Ausdruck ist, der x in der Instanzdeklaration für den Typkonstruktor c zugeordnet wird, und η'_c die Wertumgebung der Deklarationsstelle. Aufgrund der paarweisen Verschiedenheit globaler Bezeichner gilt wiederum $\eta'_c \subset \eta'_l$ und daher $\mathcal{E}'[\![M_c]\!]\eta'_c = \mathcal{E}'[\![M_c]\!]\eta'_l$. Außerdem ist $(H\eta)(x) = resolve(\omega, d)\eta$. Zu zeigen ist $(H\eta)(x) \sqsubseteq \eta'_l(x)\vartheta_l)$, was äquivalent ist zu $f \sqsubseteq g$ für $f = resolve(\omega, d)\eta$ sowie $g = resolve'(\omega, x)\vartheta$. Nun gilt $f \sqsubseteq g \iff f\overline{v_n} = g\overline{v_n}$ für alle $\overline{v_n} \in \mathbf{DV}^n$. Wir unterscheiden gemäß der Definition von *resolve* 3 Fälle: (a): $v_i = \bot$ für ein $i \in J$, (b): $\exists c : \forall i \in J : v_i \subseteq \boldsymbol{D}_c$, (c): weder (a) noch (b) treffen zu. Im Fall (a) gilt $f\overline{v_n} = \bot = g\overline{v_n}$, im Fall (c) gilt $f\overline{v_n} = wrong = g\overline{v_n}$. Im Fall (b) ist $f\overline{v_n} = \mathcal{E}[\![M_c]\!]\eta$ und $g\overline{v_n} = (((\vartheta_l x)c)\vartheta_l)\overline{v_n}$. Nun gilt aber $(((\vartheta_l x)c)\vartheta_l) = \mathcal{E}'[\![M_c]\!]\eta'_c\vartheta_l$ und da $\eta'_c \subset \eta'_l$ gilt auch $(((\vartheta_l x)c)\vartheta_l) = \mathcal{E}'[\![M_c]\!]\eta'_l\vartheta_l$. Da aber nach Induktionsvoraussetzung $\eta_l \sqsubseteq (\eta'_l, \vartheta_l)$ folgt aus Lemma 3.42 aber auch $\mathcal{E}[\![M_c]\!]\eta_l \sqsubseteq \mathcal{E}'[\![M_c]\!]\eta'_l\vartheta_l$, womit x Bedingung (ap1) erfüllt und insgesamt die Behauptung folgt. □

Damit haben wir bewiesen, daß die dynamische Semantik gemäß $\mathcal{P}'$ ein Ergebniss liefert, daß für den Wert der Fixpunktsemantik $\mathcal{P}$ eine obere Schranke darstellt. Der Autor vermutet jedoch stark, daß beide Semantikdefinition äquivalent sind, d.h. daß gilt:

$$\eta_0 \equiv (\eta'_0, \vartheta_0) \Rightarrow \mathcal{P}[\![P]\!](\eta_0, [\,]) = \mathcal{P}'[\![P]\!](\eta'_0, \vartheta_0).$$

Bedauerlicherweise kann mit der verwendeten Beweismethode nicht bewiesen werden, daß $\eta_l \equiv (\eta'_l, \vartheta_l)$ gilt: die Äquivalenz gilt nur für den Fixpunkt von H, nicht aber für seine Approximationen. Der Autor vermutet, daß man einen Beweis finden kann, der sich die Tatsache zunutze macht, daß jede Approximation $H^k(\bot_{Env})(x)$ auf sukzessiv komplexeren Argumentwerten für die überladenen Argumentpositionen definiert ist, etwa in der Form $H^k(\bot_{Env})(x)(\overline{v_n}) = \eta_l(x)\vartheta_l(\overline{v_n})$ für alle $v_i \in \mathcal{T}_o^{\bot}[\![t]\!]\emptyset$, wobei t eine zulässige Überladungsinstanz $x \uparrow^o t$ ist, für die $|t| \leq k$ gilt.

Hauptmotivation für die Betrachtung dieser zweiten Semantik ist die einfachere Übersetzbarkeit in Codesequenzen einer abstrakten Maschine. Der Ausdruck $resolve(\omega, x)$

wird in eine Fallunterscheidung übersetzt, welche die relevanten Argumentpositionen evaluiert, und dann das Typkennzeichen der Argumente benutzt, um über eine Sprungtabelle (gemäß der Operatorumgebung) die passende Instanzfunktion aufzurufen. Diese Methode ist im SAMPλE-System implementiert. Wir verzichten auf eine weitere Formalisierung.

3.5.4 Übersetzung nach Mini-ML

Im folgenden Abschnitt betrachten wir eine alternative Möglichkeit, Mini-SAMPλE-Programmen eine Bedeutung zu geben: durch *Übersetzung* nach Mini-ML. Diese Methode wurde erstmals von Wadler und Blott für das in [WB89] vorgestellte Überladungstypsystem vorgeschlagen, das in eingeschränkter Form in Haskell (siehe z.B. [HJW+91]) Eingang gefunden hat.

Wir beginnen mit einem motivierenden Beispiel: Man betrachte die Funktion *member*, mit der Definition

$$\begin{aligned} &\mathit{member} : \forall\alpha_{\{=\}}.\, \alpha_{\{=\}} \to \mathit{list}(\alpha_{\{=\}}) \to \mathit{bool} \\ &\mathit{member} = Y(\lambda f.\lambda x.\lambda l.\ \textbf{if}\ \mathit{null}\ l\ \textbf{then}\ \mathit{false}\ \textbf{else} \\ &\qquad\qquad\qquad\qquad \textbf{if}\ (\mathit{hd}\ l) = x\ \textbf{then}\ \mathit{true}\ \textbf{else}\ f(\mathit{tl}\ l)) \end{aligned}$$

Um in einem rein parametrisch polymorphen Typsystem eine vergleichbare Funktion zu erhalten, muß der Programmierer den Operator $=$ in der Definition abstrahieren

$$\begin{aligned} &\mathit{member}' : \forall\alpha.\, (\alpha \to \alpha \to \mathit{bool}) \to \alpha \to \mathit{list}(\alpha) \to \mathit{bool} \\ &\mathit{member}' = \lambda \mathit{eq}.Y(\lambda f.\lambda x.\lambda l.\ \textbf{if}\ \mathit{null}\ l\ \textbf{then}\ \mathit{false}\ \textbf{else} \\ &\qquad\qquad\qquad\qquad \textbf{if}\ \mathit{eq}\ (\mathit{hd}\ l)\ x\ \textbf{then}\ \mathit{true}\ \textbf{else}\ f(\mathit{tl}\ l)) \end{aligned}$$

und später an jeder Anwendungsstelle von member' eine Funktion übergeben, die auf Gleichheit testet. Also müßte man beispielsweise $\mathit{member}'\ \mathit{inteq}\ 7\ \langle 1, 2, 3 \rangle$ schreiben, statt einfach nur $\mathit{member}\ 7\ \langle 1, 2, 3 \rangle$.[16]

[16] Dies hat natürlich zwei Nachteile: erstens geht Typinformation und damit auch Typsicherheit verloren und zweitens ist die Konstruktion der passenden Funktion nur für elementare Datentypen trivial, man denke etwa an Listen von Mengen von endlichen Abbildungen über Basistypen!

Im allgemeinen reicht ein einziger neuer Parameter nicht aus, wie man an folgendem Beispiel sieht:

$$f : \forall\alpha_{\{=,+,*\}}.\, \alpha_{\{=,+,*\}} \to \alpha_{\{=,+,*\}} \to \alpha_{\{=,+,*\}}$$
$$f = \lambda x.\lambda y.\, \textbf{if } x = y \textbf{ then } x + y \textbf{ else } x * y$$

Für die übersetzte Funktion f' benötigt man 3 zusätzliche Parameter:

$$f' : \forall\alpha.\, (\alpha \to \alpha \to \mathit{bool}) \to (\alpha \to \alpha \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \alpha \to \alpha \to \alpha$$
$$f' = \lambda e.\lambda p.\lambda m.\lambda x.\lambda y.\, \textbf{if } e\ x\ y \textbf{ then } p\ x\ y \textbf{ else } m\ x\ y$$

Dies führt zu folgender Idee: eine Funktion f mit Typschema $\forall\overline{\alpha_n}.\tau$, wird übersetzt in eine neue Funktion f', die für jede mit einer Menge X von Operatornamen markierte Typvariable α_i, und jeden Operator aus X einen zusätzlichen Parameter erhält. Die Liste der zusätzlichen Parameter wird mit Hilfe der Funktion $\mathsf{parm} : \mathcal{V}^* \to \mathbf{Id}^*$ bestimmt.

$$\mathsf{parm}(\overline{\alpha_n}) = \mathsf{parm}(\alpha_1) \cdots \mathsf{parm}(\alpha_n) \tag{p1}$$
$$\mathsf{parm}(\alpha_{\{o_1,\ldots,o_n\}}) = \alpha_{o_1} \cdots \alpha_{o_n} \tag{p2}$$

Um eine sinnvolle Übersetzung garantieren zu können, fordern wir, daß

1. in allen Typschemata $\sigma = \forall\overline{\alpha_n}.\tau$ die generischen Variablen in einer festen Reihenfolge angegeben sind,
2. für die Operatornamen eine totale Ordnung existiert (z.B. lexikographisch),
3. die in parm verwendeten Bezeichner α_{o_i} sich von allen anderen Bezeichnern eines Programms unterscheiden.

Damit übersetzen wir dann **let** $f = M_1 \ldots$ nach **let** $f = \lambda x_1.\cdots\lambda x_m.\widehat{M_1}$, wobei $f : \forall\overline{\alpha_n}.\tau$ und $\overline{x_m} = \mathsf{parm}(\overline{\alpha_n})$ sowie $\widehat{M_1}$ den übersetzten Rumpf der Lambda-Abstraktion bezeichnet.

Durch dieses Vorgehen wird die rechte Seite der Definition jedes let-eingeführten Bezeichners um zusätzliche Parameter ergänzt. Daher müssen an jeder Verwendungsstelle eines solchen Bezeichners die passenden Funktionen als zusätzliche Parameter übergeben werden. Sei x ein solcher Bezeichner, dessen in der Typannahme zugeordnetes Typschema $\forall\overline{\alpha_n}.\tau'$ an der Verwendungsstelle über die Substitution

$S = \{\alpha_i \mapsto \tau_i\}$ zu $\tau = S\tau'$ instanziert wurde. Dann übersetzen wir x in

$$(\, x\ \mathsf{tr}(ops(\alpha_1), \tau_1)\ \ldots\ \mathsf{tr}(ops(\alpha_n), \tau_n)\,)\,,$$

wobei die (überladene) Funktion tr wie folgt definiert ist:

$$\mathsf{tr}(X, \tau) = \mathsf{tr}(o_1, \tau)\ \ldots\ \mathsf{tr}(o_n, \tau) \qquad \text{(tr1)}$$
$$\text{für } X = \{o_1, \ldots, o_n\}$$
$$\mathsf{tr}(o, \alpha_X) = \alpha_o \qquad \text{(tr2)}$$
$$\text{falls } o \in X$$
$$\mathsf{tr}(o, f(\overline{\tau_n})) = (\, o_f\ \mathsf{tr}(X_1, \tau_1)\ \ldots\ \mathsf{tr}(X_n, \tau_n)\,) \qquad \text{(tr3)}$$
$$\text{falls } o \in m(f) \wedge \forall i = 1..n : X_i = d_f(i, \{o\})$$

Für jede Instanzierung einer generischen Typvariablen α_i, die mit einer Operatormenge X markiert ist, erzeugt $\mathsf{tr}(X, \tau_i)$ eine passende Funktion. Gleichung (tr1) ist symmetrisch zur Gleichung (p2) und sorgt so dafür, daß für jeden Operator, der in X vorkommt, eine passende Funktion an x übergeben wird. $\mathsf{tr}(o, \tau)$ erzeugt einen Ausdruck, dessen Auswertung zur Laufzeit eine Funktion ergibt, welche die Semantik des Operators o für die Instanz τ eines zulässigen Argumenttyps für o liefert. Falls $\tau = \alpha_X$ mit $o \in X$, Gleichung (tr2), dann muß α_o ein Parameter eines umgebenden let-Konstrukts sein, allerdings nicht notwendigerweise des direkt übergeordneten. Die Bedingung $o \in X$ wird dabei durch die Typisierbarkeit des Programms garantiert. Gleichung (tr3) definiert, wie die einen überladenen Operator realisierende Funktion für Typkonstruktoren auf die entsprechenden Funktionen für die Argumente des Typkonstruktors zurückgeführt werden. Dabei gehen wir davon aus, daß die Übersetzung einer Instanzdeklaration eines Operators o für einen Instanztyp $f(\overline{\alpha_n})$ eine Let-Deklaration für einen neuen Bezeichner o_f erzeugt, der die notwendigen überladenen Operatoren für seine Komponententypen gemäß $d_f(i, X)$ als zusätzliche Argumente akzeptiert.

Die vollständige Übersetzung von Mini-SAMPλE nach Mini-ML definieren wir mit Hilfe eines Inferenzsystems, das in Abbildung 3.13 angegeben ist. $O, \Gamma \vdash M : \tau \xrightarrow{\mathsf{tr}} M'$ liest man am besten als: der unter der Überladungsannahme O und der Typannahme Γ mit dem Typ τ typisierbare Ausdruck M wird übersetzt in den Ausdruck M'. Die Regeln VAR und LET implementieren das o.a. Schema der zusätzlichen

Argumente an Abstraktions- und Anwendungsstellen von LET- bzw. OP- und GLB-eingeführten Bezeichnern. Deklarationen werden in eine Folge von Let-Konstrukten in Mini-ML übersetzt. Eine Operator-Erweiterung für x mit Typkonstruktor f wird gemäß Regel OV in eine Let-Deklaration für einen neuen Bezeichner x_f übersetzt. Eine Operatordeklaration mit Namen x wird in eine nicht essentielle Variablendeklaration **let** $x = ()$ überführt, um die syntaktische Korrektheit des generierten Gesamtprogramms zu gewährleisten.

Die Definition des überladenen Gleichheitsoperators auf Seite 77 wird mit diesen Regeln wie folgt übersetzt:

$$
\begin{array}{l}
\textbf{let } (=_{int}) = \mathit{inteq} \textbf{ in} \\
\textbf{let } (=_{real}) = \mathit{realeq} \textbf{ in} \\
\textbf{let } \mathit{paireq} = \lambda\alpha_{=}.\lambda\beta_{=}.\lambda p.\lambda q.\ (\alpha_{=}\ (\mathit{fst}\ p)\ (\mathit{fst}\ q)) \wedge (\beta_{=}\ (\mathit{snd}\ p)\ (\mathit{snd}\ q)) \textbf{ in} \\
\textbf{let } (=_{\times}) = \lambda\alpha_{=}.\lambda\beta_{=}.\lambda p.\lambda q.(\mathit{paireq}\ \alpha_{=}\ \beta_{=}) \textbf{ in} \\
\textbf{let } \mathit{listeq} = \lambda\alpha_{=}.Y(\lambda f.\lambda l_1.\lambda l_2.\ \textbf{if } \mathit{null}\ l_1 \textbf{ then } \mathit{null}\ l_2 \textbf{ else} \\
\qquad\qquad\qquad\qquad \neg(\mathit{null}\ l_2) \wedge (\alpha_{=}\ (\mathit{hd}\ l_1)\ (\mathit{hd}\ l_2)) \wedge f(\mathit{tl}\ l_1)\ (\mathit{tl}\ l_2)) \\
\textbf{in let } (=_{list}) = \lambda\alpha_{=}.\mathit{listeq}\ \alpha_{=} \textbf{ in} \\
\textbf{eval } (=_{list}\ (=_{\times}\ (=_{real})\ (=_{int}))\ \langle(1.0, 2)\rangle\ \langle\,\rangle\ \langle\,\rangle
\end{array}
$$

Die Übersetzung beruht auf der Tatsache, daß man Applikationen von überladenen Funktionen anhand ihres Typs auflösen kann, entweder dadurch, daß an der Anwendungsstelle der Typ monomorph und somit die notwendige Instanz eindeutig bestimmt ist, oder daß der überladene Operator abstrahiert werden kann und somit an einer späteren Verwendungsstelle der umgebenden Funktion bekannt wird und dort aufgelöst werden kann. Es gibt jedoch Vorkommen von überladenen Operatoren, die so nicht aufgelöst werden können, wie man an folgendem Beispiel sieht:

$$\textbf{let } x = \lambda y.\, \textbf{if } \langle\,\rangle = \langle\,\rangle \textbf{ then } 1 \textbf{ else } y\ ;\ \ldots$$

Der allgemeinste Typ der Funktion x ist $\mathit{int} \to \mathit{int}$, der Vergleich der beiden Listen dabei mit $\alpha_{\{=\}} \to \alpha_{\{=\}} \to \mathit{bool}$ typisiert. Kein Kontext, indem x verwendet werden kann, wird je dazu führen, daß der Typ der Listenelemente bekannt wird, sodaß die Überladungsinstanz von $=$ niemals bestimmt werden kann. Übersetzung gemäß $\xrightarrow{\mathsf{tr}}$ liefert

$$\textbf{let } x = \lambda y.\, \textbf{if } (=_{list}\ \ \alpha_{=})\ \langle\,\rangle\ \langle\,\rangle \textbf{ then } 1 \textbf{ else } y\ ;\ \ldots$$

[VAR] $$\frac{\Gamma(x) = \forall\overline{\alpha_n}.\tau' \quad \tau = \{\alpha_i \mapsto \tau_i\}\tau'}{O, \Gamma \vdash x : \tau \xrightarrow{\mathsf{tr}} (\, x\ \mathsf{tr}(ops(\alpha_1), \tau_1)\ \ \ldots\ \ \mathsf{tr}(ops(\alpha_n), \tau_n)\,)}$$

[ABS] $$\frac{O, \Gamma + [x : \tau] \vdash M : \tau' \xrightarrow{\mathsf{tr}} M'}{O, \Gamma \vdash \lambda x.M : \tau \to \tau' \xrightarrow{\mathsf{tr}} \lambda x.M'}$$

[APP] $$\frac{O, \Gamma \vdash M_1 : \tau \to \tau' \xrightarrow{\mathsf{tr}} M_1' \qquad O, \Gamma \vdash M_2 : \tau \xrightarrow{\mathsf{tr}} M_2'}{\Gamma \vdash M_1 M_2 : \tau' \xrightarrow{\mathsf{tr}} M_1' M_2'}$$

[LET] $$\frac{\begin{array}{c} O, \Gamma \vdash M_1 : \tau \xrightarrow{\mathsf{tr}} M_1' \qquad \sigma = \mathrm{gen}(\Gamma, \tau) = \forall\overline{\alpha_n}.\tau \\ O, \Gamma + [x : \sigma] \vdash M_2 : \tau' \xrightarrow{\mathsf{tr}} M_2' \qquad \overline{\gamma_k} = \mathsf{parm}(\overline{\alpha_n}) \end{array}}{O, \Gamma \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau' \xrightarrow{\mathsf{tr}} \mathbf{let}\ x = \lambda\gamma_1.\cdots\lambda\gamma_k.\ M_1'\ \mathbf{in}\ M_2'}$$

[OP] $$\frac{\sigma = \forall\alpha_{\{x\}}.\{\$ \to \alpha_{\{x\}}\}\omega \qquad x \notin dom(O) \quad x \notin dom(\Gamma)}{O, \Gamma \vdash^d \mathbf{operator}\ x :: \omega : O + [x : \langle\omega, []\rangle], \Gamma + [x : \sigma] \xrightarrow{\mathsf{tr}} \mathbf{let}\ x = ()}$$

[OV] $$\frac{\begin{array}{c} \sigma = \mathrm{gen}(\Gamma, \tau) = \forall\overline{\alpha_n}.\{\$ \to f(\overline{\alpha_n})\}\omega \\ O(x) = \langle\omega, d\rangle \wedge d' = d \cup \{f(\overline{\alpha_n})\} \\ O, \Gamma \vdash^e M : \tau \xrightarrow{\mathsf{tr}} M' \qquad \nexists f(\overline{\beta_n}) \in d \qquad \overline{\gamma_k} = \mathsf{parm}(\overline{\alpha_n}) \end{array}}{\begin{array}{l} O, \Gamma \vdash^d \mathbf{extend}\ x :: \sigma\ \mathbf{with}\ M : O + [x : \langle\omega, d'\rangle], \Gamma \\ \xrightarrow{\mathsf{tr}} \mathbf{let}\ x_f = \lambda\gamma_1.\cdots\lambda\gamma_k.\ M' \end{array}}$$

[GLB] $$\frac{\begin{array}{c} O, \Gamma \vdash^e M : \tau \xrightarrow{\mathsf{tr}} M' \\ \sigma = \mathrm{gen}(\Gamma, \tau) = \forall\overline{\alpha_n}.\tau \qquad \overline{\gamma_k} = \mathsf{parm}(\overline{\alpha_n}) \end{array}}{O, \Gamma \vdash^e \mathbf{let}\ x = M_1 : O, \Gamma + [x : \sigma] \xrightarrow{\mathsf{tr}} \mathbf{let}\ x = \lambda\gamma_1.\cdots\lambda\gamma_k.\ M'}$$

[SEQ] $$\frac{O, \Gamma \vdash^d D : O', \Gamma' \xrightarrow{\mathsf{tr}} D' \qquad O', \Gamma' \vdash^p P \xrightarrow{\mathsf{tr}} P'}{O, \Gamma \vdash^p D; P \xrightarrow{\mathsf{tr}} D'\ \mathbf{in}\ P'}$$

[VAL] $$\frac{O, \Gamma \vdash^e M : \tau \xrightarrow{\mathsf{tr}} M'}{O, \Gamma \vdash^p \mathbf{eval}\ M \xrightarrow{\mathsf{tr}} M'}$$

Abbildung 3.13: Übersetzung von Mini-SAMPλE nach Mini-ML

$$[\text{VAR}] \quad \frac{\tau \prec^o \Gamma(x)}{\Gamma \vdash^o x : \tau \xrightarrow{\text{ftv}} \mathcal{V}(\tau) - \mathcal{FV}(\Gamma)}$$

$$[\text{ABS}] \quad \frac{\Gamma + [x : \tau] \vdash^o M : \tau' \xrightarrow{\text{ftv}} V}{\Gamma \vdash^o \lambda x.M : \tau \to \tau' \xrightarrow{\text{ftv}} V \cup (\mathcal{V}(\tau) - \mathcal{FV}(\Gamma))}$$

$$[\text{APP}] \quad \frac{\Gamma \vdash^o M_1 : \tau \to \tau' \xrightarrow{\text{ftv}} V_1 \ , \ \Gamma \vdash^o M_2 : \tau \xrightarrow{\text{ftv}} V_2}{\Gamma \vdash^o M_1 M_2 : \tau' \xrightarrow{\text{ftv}} V_1 \cup V_2}$$

$$[\text{LET}] \quad \frac{\Gamma \vdash^o M_1 : \tau \xrightarrow{\text{ftv}} V_1 \ , \ \Gamma + [x : \text{gen}(\Gamma, \tau)] \vdash^o M_2 : \tau' \xrightarrow{\text{ftv}} V_2}{\Gamma \vdash^o \textbf{let } x = M_1 \textbf{ in } M_2 : \tau' \xrightarrow{\text{ftv}} (V_1 - \mathcal{BV}(\text{gen}(\Gamma, \tau))) \cup V_2}$$

Abbildung 3.14: Deduktion freier Typvariablen

d.h. das erzeugt Programm M' enthält eine freie Variable $\alpha_=$, die im Originalprogramm nicht vorhanden war und somit hat das erzeugte Programm weder eine Semantik, noch kann es als Mini-ML-Programm typisiert werden. Dies ist natürlich sehr bedauerlich, da das Programm gemäß der denotationalen Semantikdefinition eine klar definierte Semantik besitzt![17]

Hier stellt sich natürlich die Frage, ob man die Übersetzungsfunktion so verbessern kann, daß das obige Beispiel übersetzbar wird. Betrachtet man es etwas genauer, dann sieht man, daß die Vergleichsfunktion für Listenelemente niemals benötigt wird:

$$\mathit{listeq}\ f\ \langle\,\rangle\ \langle\,\rangle = \mathit{true}$$

für beliebige Funktionen f. Also könnte man für diesen Fall einfach die Typvariable $\alpha_{\{=\}}$ im Originalprogramm durch eine beliebige Instanz ersetzen, z.B. *int*, womit die freie Variable im generierten Mini-ML-Code eliminiert wird:

$$\textbf{let } x = \lambda y.\, \textbf{if } (=_{\mathit{list}}\ \ (=_{\mathit{int}}))\ \langle\,\rangle\ \langle\,\rangle \textbf{ then } 1 \textbf{ else } y\ ;\ \ldots$$

[17] Bedauerlich ist auch, daß man am Typ von x das Problem nicht erkennen kann. Dies wird erst durch den Übergang zur Typdeduktion mit Restriktionen gelöst. Dort bekommt x den allgemeinsten Typ $\mathit{int} \to \mathit{int}|\alpha_{\{=\}}$ zugeordnet (siehe Kapitel 4 und auch [NP93]).

Das Problem mit diesem Ansatz ist jedoch, daß es u.U. keine Instanz für die Typvariable α_X gibt, sodaß $\tau \uparrow^o X$ gilt. Besser ist es daher, einfach im generierten Code, alle freien Variablen, die durch die Übersetzung generiert wurden, durch die überall undefinierte Funktion $f_\perp$ zu ersetzen. Man erhält somit für das betrachtete Beispiel den Ausdruck

$$\textbf{let } x = \lambda y.\, \textbf{if } (=_{list}\ \ f_\perp)\ \langle\rangle\ \langle\rangle \textbf{ then } 1 \textbf{ else } y\ ;\ \ldots$$

Die Funktion $f_\perp$ ist definiert durch $f_\perp\, x = \perp$ und besitzt das Typschema $\forall\alpha, \beta.\alpha \to \beta$. Damit ist klar, daß das generierte Programm ML-typisierbar bleibt; für die Semantik vermutet der Autor:

Vermutung 3.44. *Im Falle der laufzeitauflösbaren, parametrischen Überladungen ist das Ersetzen aller durch die Übersetzung generierten freien Variablen durch $f_\perp$ möglich, ohne daß die Semantik des generierten Programms von der denotationalen Semantik des Original-Programms abweicht.*

Es lassen sich leicht weitere Beispiele finden, für die das Vorgehen korrekt ist. So kann etwa das obige Beispiel direkt auf alle Fälle übertragen werden, bei denen die leere Liste durch entsprechende Konstanten anderer polymorpher Datenstrukturen ersetzt wird: leere Bäume, leere Mengen, leere endliche Abbildungen, usw. Auch nicht terminierende Berechnungen geben gute Beispiele ab:

$$\textbf{let } x = \mathit{fst}\ (\,1, Y(\lambda x.x + x)\,)\ ;\ \ldots$$

Hier hat x den Typ *int* und der Unterausdruck $M = Y(\lambda x.x + x)$ den Typ $\alpha_{\{+\}}$. Jedoch ist die Bedeutung von M klarerweise $\perp$ und wir können das generierte Programm in

$$\textbf{let } x = \mathit{fst}\ (\,1, Y(\lambda x.f_\perp\ x\ x)\,)\ ;\ \ldots$$

abändern, ohne daß die Semantik geändert wird.

Die Situation ist ähnlich zu dem Problem der „tag free garbage collection“, bei der man versucht, den Typ der Datenstrukturen auf der Halde ausgehend von den Typen der aktuellen Parameter von Aufrufen polymorpher Funktionen zu bestimmen (siehe [App89, Gol91] für die zugrundeliegenden Ideen). Dort können jedoch zur Laufzeit auf der Halde Speicherstrukturen entstehen, für die der Typ aus den Argumenttypen

des Funktionsaufrufs nicht mehr ermittelt werden kann.[18] Man kann nun beweisen, daß in diesem Fall nie mehr auf die entsprechenden Werte zugegriffen werden muß, und man somit auf das Kollektieren derselben verzichten kann (siehe [GG92]). Daher kann man hoffen, daß sich der dortige Beweis anpassen läßt, um obige Vermutung zu bestätigen.

Die Vermutung gilt jedoch nur für die laufzeitauflösbaren Überladungen, die wir bisher betrachtet haben. Verzichtet man auf die Forderung, daß die Überladungsschemata ω von der Form $\omega_0 \to \cdots \to \omega_n$ sein müssen, wobei $\omega_i = \$$ für ein $i < n$, und erlaubt beliebige Typausdrücke über $T_F(\mathcal{V}^{\$})$ als Überladungsschemata, dann bleiben zwar sämtliche Typisierbarkeits-/Typinferenzresultate erhalten, aber die denotationale Semantik kann so nicht weiter verwendet werden und es bleibt nur die Übersetzung als Definition der Semantik. Diese Art der Überladungen bezeichnen wir daher als *nicht laufzeitauflösbare parametrische Überladungen.*

Beispiele für solche Überladungen sind zum einen überladene Konstanten. So könnte man eine Konstante *zero* definieren, die mit *int* und *real* als Instanzen überladen werden:

$$\begin{array}{l}\textbf{operator}\ \mathit{zero} :: \$\\ \textbf{extend}\ \mathit{zero} :: \mathit{int}\ \textbf{with}\ 0\\ \textbf{extend}\ \mathit{zero} :: \mathit{real}\ \textbf{with}\ 0.0\end{array}$$

Dies hätte den Vorteil, z.B. eine allgemeine Funktion zur Addition einer möglicherweise leeren Liste von Zahlen definieren zu können:

$$\begin{array}{l}\textbf{let}\ \mathit{add} = Y(\lambda f.\lambda l.\ \textbf{if}\ \mathit{null}\ l\ \textbf{then}\ \mathit{zero}\ \textbf{else}\ \mathit{hd}\ l + f(\mathit{tl}\ l))\\ \textbf{eval}\ \mathit{add}\ \langle\, 1, 2, 3\,\rangle\end{array}$$

Die Funktion *add* besitzt den Typ $\forall\alpha_{\{+,zero\}}.\mathit{list}(\alpha_{\{+,zero\}}) \to \alpha_{\{+,zero\}}$. Zum anderen können nun Operatoren definiert werden, deren Ergebnis nicht vom Typ der Argumente abhängt. Typisches Beispiel dafür ist ein Operator *read*, der einen String

[18] Diese Beobachtung resultierte aus einer Untersuchung aktueller Garbage-Collector-Algorithmen für den Einsatz in SAMPλE.

parsen und in einen Wert des Instanztyps umwandeln soll:

$$\begin{aligned}
&\textbf{operator } \mathit{read} :: \mathit{list}(\mathit{char}) \to \$\\
&\textbf{extend } \mathit{read} :: \mathit{list}(\mathit{char}) \to \mathit{int} \textbf{ with } \mathit{parseint}\\
&\textbf{extend } \mathit{read} :: \mathit{list}(\mathit{char}) \to \mathit{real} \textbf{ with } \mathit{parsereal}\\
&\textbf{extend } \mathit{read} :: \mathit{list}(\mathit{char}) \to \mathit{list}(\alpha_{\{\mathit{read}\}}) \textbf{ with } \ldots
\end{aligned}$$

Dessen Gegenstück *show* ist im SAMPλE-System verfügbar:

$$\begin{aligned}
&\textbf{operator } \mathit{show} :: \$ \to \mathit{list}(\mathit{char}) \to \mathit{list}(\mathit{char})\\
&\textbf{extend } \mathit{show} :: \mathit{int} \to \mathit{list}(\mathit{char}) \to \mathit{list}(\mathit{char}) \textbf{ with } \mathit{showint}\\
&\textbf{extend } \mathit{show} :: \mathit{real} \to \mathit{list}(\mathit{char}) \to \mathit{list}(\mathit{char}) \textbf{ with } \mathit{showreal}\\
&\textbf{extend } \mathit{show} :: \mathit{list}(\alpha_{\{\mathit{show}\}}) \to \mathit{list}(\mathit{char}) \to \mathit{list}(\mathit{char}) \textbf{ with } \ldots
\end{aligned}$$

Dabei gilt $\mathit{read}(\mathit{show}\, s\, \langle\rangle) = s$.

Ein weiteres interessantes Beispiel liefert die Behandlung von Konstanten in Haskell. Dort wird jedes Vorkommen einer ganzzahligen[19] Konstanten im Quelltext als impliziter Aufruf eines überladenen Operators $\mathit{fromInteger} : \mathit{Integer} \to \$$ verstanden, der für alle Zahltypen überladen ist.[20] Es ist klar, daß dadurch die Wahrscheinlichkeit von nicht auflösbaren Überladungen stark erhöht wird und daher wurde für Haskell festgelegt, daß bei nicht auflösbaren Überladungen, die durch *fromInteger* induziert werden, eine *voreingestellte*, automatische Instanzierung durch den Typ *Int* vorgenommen wird.

Der parametrisch überladene Typ eines Ausdrucks im Originalprogramm und der parametrisch polymorphe Typ des generierten Programms stehen in einem systematischen Zusammenhang, der über eine Funktion ml angegeben werden kann, die überladene Typen in ML-Typen abbildet:

$$\begin{aligned}
\mathsf{ml}(\forall\overline{\alpha_n}.\tau) &= \forall\overline{S(\alpha_k)}.\mathsf{ml}(\alpha_1,\beta_1) \to \cdots \to \mathsf{ml}(\alpha_k,\beta_k) \to S\tau\\
&\quad \text{wobei } S = \{\alpha_i \mapsto \beta_i \mid \mathit{ops}(\alpha_i) \neq \emptyset\}\\
&\quad \text{für } k = |\{\alpha_i \mid \mathit{ops}(\alpha_i) \neq \emptyset\}| \text{ neue Typvariablen}\\
\mathsf{ml}(\alpha_{\{o_1,\ldots,o_n\}},\beta) &= \{\$ \mapsto \beta\}(\omega_1 \to \cdots \to \omega_n)\\
&\quad \text{wobei } o_i : \langle\omega_i, s\rangle \in O \text{ für } i = 1..n
\end{aligned}$$

[19] Für Fließkommazahlen gilt eine ähnliche Regel.

[20] und natürlich vom Anwender beliebig erweitert werden kann.

Vermutung 3.45. *Sei* $O, \Gamma \vdash M : \tau$ *eine gültige Überladungstypisierung und* $O, \Gamma \vdash M : \tau \xrightarrow{\mathsf{tr}} M'$ *die Übersetzung von* M*. Es gelte* $O, \Gamma \vdash M : \tau \xrightarrow{\mathsf{ftv}} V$ *mit* $V \subseteq \mathcal{V}^{\emptyset}$*. Dann ist* $\mathsf{ml}(\Gamma) \vdash \lambda\gamma_1.\cdots\lambda\gamma_k.M' : \tau'$ *eine gültige Typisierung, wobei* $\forall\overline{\alpha_n}.\tau = \mathrm{gen}(\Gamma, \tau)$ *und* $\mathsf{ml}(\forall\overline{\alpha_n}.\tau) = \forall\beta_m.\tau'$ *und* $\overline{\gamma_k} = \mathsf{parm}(\overline{\alpha_n})$*.* [21]

3.6 Typklassen

In diesem Abschnitt zeigen wir, daß das aus der funktionalen Programmiersprache Haskell bekannte Konzept der Typklassen und das vom Autor definierte Konzept der parametrischen Überladungen in einem engen Zusammenhang stehen. Dazu zeigen zunächst, daß man durch eine minimale Modifikation der Syntax von Überladungsschemata sofort ein Typdeduktionssystem und einen Typinferenzalgorithmus für Typklassen erhält. Anschließend definieren wir eine Übersetzung von Mini-Haskell nach Mini-SAMPλE.

3.6.1 Mini-Haskell

Die grundlegende Idee des Haskell-Typsystems, die erstmals in [WB89] vorgestellt wurde, ist die Gruppierung von verwandten überladenen Operatoren in sogenannte *Typklassen*. Die vollständige Syntax einer vereinfachten Teilmenge von Haskell ist in Abbildung 3.15 angegeben.[22] Für die umfassende Haskell-Definition verweisen wir den Leser auf einen Haskell-Report, z.B. [HJW+91].

Eine Klassendeklaration der Form **class** $K(\alpha) \Rightarrow C(\alpha)$ **where** $x_1 : \tau_1 \ldots x_n : \tau_n$ deklariert eine neue Typklasse C, mit n überladenen Operatoren $\overline{x_n}$, sowie entsprechenden Typschemata $\overline{\tau_n}$. Dabei werden alle Typvariablen aus $\mathcal{V}(\overline{\tau_n})$ implizit quantifiziert.

Polymorphe Typausdrücke haben in Haskell die Form $K \Rightarrow \tau$, wobei $\emptyset \Rightarrow \tau$ einfach durch τ notiert wird. Einem Operator $x : \tau$ einer Klassendeklaration $C(\alpha)$ wird der Typausdruck $C(\alpha) \Rightarrow \tau$ zugeordnet. Somit erhalten $=$ und $\neq$ aus der Klasse Eq beide den polymorphen Typausdruck $Eq(\alpha) \Rightarrow \alpha \to \alpha \to \mathit{bool}$ (siehe Abbildung 3.16).

[21] Wir verzichten an dieser Stelle auf einen Satz und den zugehörigen formalen Beweis, da eine ähnliche Aussage auch für die in [NS91] vorgestellte Übersetzung für Mini-Haskell getroffen wurde.

[22] $[s]$ bezeichnet ein optionales syntaktisches Konstrukt s.

Ausdrücke	M	$::=$	$x \mid \lambda x.M \mid M_1 M_2$
		$\mid$	$\textbf{let}\ x = M_1\ \textbf{in}\ M_2$
Deklarationen	D	$::=$	$\textbf{class}\ K(\alpha) \Rightarrow C(\alpha)\ \textbf{where}$
			$x_1 : \tau_1 \ \ldots\ x_n : \tau_n$
		$\mid$	$\textbf{instance}\ K(\overline{\alpha_k}) \Rightarrow C(f(\overline{\alpha_k}))\ \textbf{where}$
			$x_1 = e_1 \ \ldots\ x_n = e_n$
			wobei $k = \rho(f), i \neq j \Rightarrow \alpha_i \neq \alpha_j$
		$\mid$	$\textbf{let}\ x = M$
Klassenkontexte	$K(\overline{\alpha_n})$	$::=$	$C_1(\alpha_{i_1}), \ldots, C_k(\alpha_{i_k})$ für $i_j \in \{1..n\}$
Programme	P	$::=$	$D; P \mid \textbf{eval}\ M$

Abbildung 3.15: Syntax von Mini-Haskell

Über den Klassenkontext $K(\alpha)$ wird darüber hinaus C als Unterklasse aller in $K(\alpha)$ aufgeführten Klassen deklariert, wobei die entstehende Klassenhierarchie azyklisch sein muß. Auf den Deklarationstyp der in C deklarierten Operatoren hat dies jedoch keinen Einfluß: es bedeutet lediglich, daß eine Instanzdeklaration $C(f(\overline{\alpha_k}))$ nur erlaubt ist, wenn f schon als Instanz für alle Oberklassen von C deklariert wurde. Darüber hinaus erlaubt die Unterklassenbeziehung die Vereinfachung von Typausdrücken: falls C_2 eine Oberklasse von C_1 ist, dann ist der Typausdruck $C_1(\alpha), C_2(\alpha) \Rightarrow \tau$ äquivalent zu dem Ausdruck $C_1(\alpha) \Rightarrow \tau$, da jede Instanz von C_1 auch eine Instanz von C_2 sein muß. Beispiel: $\mathit{Num}(\alpha), \mathit{Eq}(\alpha) \Rightarrow \alpha \to \alpha$ ist äquivalent zu $\mathit{Num}(\alpha) \Rightarrow \alpha \to \alpha$, da Eq eine Oberklasse von Num ist. Im Folgenden schreiben wir $\leq$ für die reflexive und transitive Hülle der Oberklassenrelation: $C_1 \leq C_2$ falls C_1 eine Oberklasse von C_2 ist (also $\mathit{Eq} \leq \mathit{Num}$).

Über eine Instanzdeklaration der Form $\textbf{instance}\ K(\overline{\alpha_k}) \Rightarrow C(f(\overline{\alpha_k}))\ \textbf{where}\ x_1 = e_1 \ \ldots\ x_n = e_n$ wird die Menge der zulässigen Instanzen der Klasse C um alle Typen $f(\overline{\tau_k})$ erweitert, für die die τ_i Instanzen aller im Klassenkontext K aufgeführten Klassen sind. Dies entspricht der induktiven Definition der zulässigen Argumenttypen für überladene Operatoren in Mini-SAMPλE. Beispiel: die Instanzdeklaration $\textbf{instance}\ \mathit{Eq}(\alpha) \Rightarrow \mathit{Eq}(\mathit{list}(\alpha)) \ldots$ führt Gleichheit auf Listen zurück auf Gleichheit

---

class $Eq(\alpha)$ **where**
$(=) : \alpha \to \alpha \to bool$
$(\neq) : \alpha \to \alpha \to bool$

instance $Eq(int)$ **where**
$(=) = inteq$
$(\neq) = intneq$

instance $Eq(\alpha) \Rightarrow Eq(list(\alpha))$ **where**
$(=) = listeq$
$(\neq) = not \circ listeq$

class $Eq(\alpha) \Rightarrow Num(\alpha)$ **where**
$(+) : \alpha \to \alpha \to \alpha$
$(*) : \alpha \to \alpha \to \alpha$

instance $Num(int)$ **where**
$(+) = intadd$
$(*) = intmult$

Abbildung 3.16: Typklassen-Beispiele

der Listenelemente. *listeq* hat dabei den Typ $Eq(\alpha) \Rightarrow \alpha \to \alpha \to bool$.

3.6.2 Typdeduktion und Typinferenz für Mini-Haskell

Man sieht leicht, daß sich ein Typausdruck $K \Rightarrow \tau$ äquivalent durch einen Typausdruck über mit Klassennnamen sortierten Typvariablen darstellen läßt. Für $Eq(\alpha) \Rightarrow \alpha \to \alpha \to bool$ erhält man so $\alpha_{\{Eq\}} \to \alpha_{\{Eq\}} \to bool$. Allgemein definieren wir für K eine Substitution S_K mit $S_K(\alpha) = \alpha_{\{C' | C(\alpha) \in K \wedge C' \leq C\}}$, sodaß $K \Rightarrow \tau$ durch $S_K(\tau)$ angemessen repräsentiert wird.

Definition 3.46 (Erweiterte Typschemata und Überladungsannahmen). Ein erweitertes Typschema ist eine endliche Abbildung von Operatornamen auf einfache Typschemata. Eine erweiterte Überladungsannahme O ist eine endliche Abbildung von Klassennamen auf Paare $\langle \omega, s \rangle$, wobei alle τ in s von der Form $f(\alpha_n)$ für $f \in F_n$.

Für eine gegebene Menge M von Klassen und Instanzdeklarationen kann man systematisch eine erweiterte Überladungsannahme O_M konstruieren, mit $O_M(C) = \langle m_C, s_C \rangle$ und

$$\begin{aligned} m_C &= \{\alpha \mapsto \$\}\overline{x_n : \tau_n} \qquad \text{wobei } \mathbf{class}\ C(\alpha)\ \mathbf{where}\ \overline{x_n : \tau_n} \in M \\ s_C &= \{S_K(f(\overline{\alpha_k})) \mid \mathbf{instance}\ K \Rightarrow C(f(\overline{\alpha_k}))\ \mathbf{where}\ \overline{x_n = \tau_n} \in M\} \end{aligned}$$

Offensichtlich ist τ eine Instanz der Klasse C, genau dann, wenn $C \uparrow^{O_M} \tau$ erfüllt ist und somit erhält man sofort ein Typdeduktionssystem für Mini-Haskell. Da außerdem der Unifikationsalgorithmus für überladungssortierte Typausdrücke nicht von der Syntax der Überladungsschemata abhängt, erhält man auch einen Typinferenzalgorithmus für Mini-Haskell.

Allerdings werden von diesem Algorithmus die Oberklassenbeziehungen noch nicht zur Vereinfachung der inferierten Typen verwendet. Dazu definieren wir kurzerhand eine Äquivalenzrelation auf Mengen von Typklassen:

$$X_1 \equiv X_2 \iff lb(X_1) = lb(X_2) \qquad \text{wobei} \quad lb(X) \stackrel{\text{def}}{=} \{C' \mid \exists C \in X : C' \leq C\}$$

Jede Äquivalenzklasse hat einen eindeutig bestimmten Repräsentanten, der aus der Menge der maximalen Elemente bzgl. der Unterklassenbeziehung besteht. Damit können wir entweder den Unifikationsalgorithmus so abändern, daß er nur mit den Repräsentanten arbeitet, oder aber nach Abschluß der Typinferenzphase[23] in allen inferierten Typausdrücken die Sorten der Typvariablen durch die entsprechenden Repräsentanten ersetzen.

[23] Genauso kann man vorgehen für Fehlermeldungen, die dem Nutzer während der Typinferenzphase präsentiert werden.

---

operator $(=) : \$ \to \$ \to \mathit{bool}$
operator $(\neq) : \$ \to \$ \to \mathit{bool}$

extend $(=) : \mathit{int} \to \mathit{int} \to \mathit{bool}$ **with** *inteq*
extend $(\neq) : \mathit{int} \to \mathit{int} \to \mathit{bool}$ **with** *intneq*

extend $(=) : \forall\alpha_{\{=,\neq\}}.\mathit{list}(\alpha_{\{=,\neq\}}) \to \mathit{list}(\alpha_{\{=,\neq\}}) \to \mathit{bool}$ **with** *listeq*
extend $(\neq) : \forall\alpha_{\{=,\neq\}}.\mathit{list}(\alpha_{\{=,\neq\}}) \to \mathit{list}(\alpha_{\{=,\neq\}}) \to \mathit{bool}$ **with** $\mathit{not} \circ \mathit{listeq}$

operator $(+) : \$ \to \$ \to \$$
operator $(*) : \$ \to \$ \to \$$

extend $(+) : \mathit{int} \to \mathit{int} \to \mathit{int}$ **with** *intadd*
extend $(*) : \mathit{real} \to \mathit{real} \to \mathit{real}$ **with** *intmult*

Abbildung 3.17: Nach Mini-SAMPλE übersetzte Typklassenbeispiele

3.6.3 Übersetzung von Mini-Haskell nach Mini-SAMPλE

Die Übersetzung von Mini-Haskell nach Mini-SAMPλE liefert ein Mini-SAMPλE Programm, das genau dann typisierbar ist, wenn das Original-Programm typisierbar war. Wir geben nur eine informelle Beschreibung.

Eine Klassendeklaration der Form

$$\textbf{class } K(\alpha) \Rightarrow C(\alpha) \textbf{ where } x_1 : \tau_1 \ \ldots \ x_n : \tau_n$$

wird in in eine Folge von Operator-Deklarationen in Mini-SAMPλE übersetzt:

$$\textbf{operator } x_1 : \mathrm{gen}(\emptyset, \{\alpha \mapsto \$\}\tau_1) ; \ldots ; \textbf{operator } x_n : \mathrm{gen}(\emptyset, \{\alpha \mapsto \$\}\tau_n)$$

wobei sich das Überladungsschema eines Operators durch Ersetzen der ausgezeichneten Klassenvariablen α durch \$ und anschließender Abstraktion aller freien Typvariablen ergibt.

Eine Instanzdeklaration der Form

$$\textbf{instance } K(\overline{\alpha_k}) \Rightarrow C(f(\overline{\alpha_k})) \textbf{ where } x_1 = e_1 \ \ldots \ x_n = e_n$$

wird übersetzt in eine entsprechende Reihe von **extend** Klauseln in Mini-SAMPλE:

$$\begin{array}{l}\textbf{extend } x_1 : \mathrm{gen}(\emptyset, \{\alpha \mapsto S^*_K(f(\overline{\alpha_k}))\}(\Gamma_C(x_1))) \textbf{ with } e_1 \\ \vdots \\ \textbf{extend } x_n : \mathrm{gen}(\emptyset, \{\alpha \mapsto S^*_K(f(\overline{\alpha_k}))\}(\Gamma_C(x_n))) \textbf{ with } e_n\end{array}$$

wobei wir mit Γ_C die Typannahme bezeichnen, die jedem überladenen Operator x aus C seinen durch die Übersetzung entstandenen Deklarationstyp zuordnet und $S^*_K(\alpha) = \alpha_{\{x|C(\alpha)\in K \wedge C' \leq C \wedge x \in dom(\Gamma_{C'})\}}$. Man beachte, daß aufgrund der Haskell-Typisierungsregeln die Reihenfolge der generierten Extend-Klauseln irrelevant ist. Abbildung 3.17 enthält den generierten Code für die obigen Typklassen-Beispiele.

Durch die Übersetzung geht die Information über die Klassenhierarchie verloren. Sie ist nur noch implizit durch die Teilmengenbeziehung zwischen Operatormengen vorhanden. Dafür sind jedoch die inferierten allgemeinsten Typen „genauer“: die inferierten Typen geben Auskunft, welche der überladenen Operatoren einer Typklasse in der Funktion nach außen sichtbar verwendet werden. Beispielsweise hat die Funktion $\lambda x.x+x$ im Originalprogramm den allgemeinsten Typ $\forall\alpha.Num(\alpha) \Rightarrow \alpha \to \alpha$, im übersetzten Programm lautet der allgemeinste Typ dagegen: $\forall\alpha_{\{+\}}.\alpha_{\{+\}} \to \alpha_{\{+\}}$.

3.7 Diskussion

Die vom Autor entwickelte Theorie der parametrischen Überladungen zeigt, daß sich auch der sogenannte „ad hoc“ Polymorphismus systematisch behandeln läßt. Dabei wurden nicht nur theoretische Resultate erzielt: das Typkonzept der vom Autor im Rahmen des DFG-Forschungsprojekts „Umgebung für konstruktive Spezifikationen“ mitentwickelten Spezifikations- bzw. Programmiersprache SAMPλE basiert in wesentlichen Teilen auf der Theorie der parametrischen Überladungen. Sprachbeschreibungen findet man in [Jäg87, Jäg88b, JGK88] bzw. [KG89] und im Anhang der Dissertation von Michael Gloger [Glo92]. Eine Beschreibung der Implementierung der Typinferenz für benutzerdefinierbare Überladungen findet man in [Oph91].

Die SAMPλE-Programmierumgebung [Jäg88a, GKT90] wurde über mehrere Jahre in Lehre und Forschung des Fachgebietes Praktische Informatik der TU-Darmstadt eingesetzt.

Die wichtigsten Resultate der Theorie der parametrischen Überladungen fassen wir an dieser Stelle noch einmal kurz zusammen:

1. Typisierbarkeit ist entscheidbar.

2. Parametrische Überladungen erhalten die „principal type property“ des parametrischen Polymorphismus.

3. Der allgemeinste Typ eines Ausdrucks kann durch einen einzigen Typausdruck dargestellt werden.

4. Typinferenz basiert auf einem unitären Unifikationsalgorithmus, der Überladungen berücksichtigt und der polynomial zeitbeschränkt implementierbar ist.

5. Wohltypisierte Programme erzeugen keine Typfehler zur Laufzeit.

6. Kontextunabhängige Überladungen ermöglichen die Definition einer denotationalen Semantik, die nicht von der konkreten Typisierung eines Ausdrucks abhängt.

7. Parametrische Überladungen erlauben auch kontextabhängige Überladungen, diese erzwingen allerdings eine Definition der Semantik durch Übersetzung in eine Sprache ohne Überladungen und beinhalten das Problem unauflösbarer Überladungen.

8. Kontextunabhängige Überladungen können bei einer Übersetzung in eine überladungsfreie Sprache vollständig aufgelöst werden.

9. Parametrische Überladungen und Haskell-Typklassen sind in etwa gleich mächtige Typisierungskonzepte.

10. Die Instanzrelation von Typausdrücken ist entscheidbar in Polynomialzeit.

11. Die Frage, ob ein gegebener, überladener Typausdruck eine überladungsfreie Instanz hat, ist entscheidbar, aber für die Frage der Typisierbarkeit und der Semantik ohne Relevanz.

Der auffallendste Unterschied zwischen dem SAMPλE-Typsystem und dem Haskell-Typsystem ist die Beschränkung auf laufzeitauflösbare Überladungen im ersten, im Gegensatz zu kontextabhängigen im zweiten. Beide Methoden haben Vor- und Nachteile: laufzeitauflösbare Überladungen bedingen, daß Konstanten nicht überladen werden können und abstrakte Datentypen mit einem eindeutigen Typkennzeichen versehen werden müssen, das die Verzweigung in die zu dem Instanztyp gehörende Funktion über eine Sprungtabelle steuert. Das Kennzeichen muß beim Anlegen der Daten auf dem Heap geschrieben und beim Zugriff entsprechend dereferenziert werden und erzeugt damit zusätzlichen Aufwand im Laufzeitsystem.[24] Andererseits ist die Übersetzung für laufzeitauflösbare Überladungen relativ einfach und die Kennzeichen vereinfachen die Implementierung eines aussagekräftigen Debuggingsystems auf Basis des Boxmodells (siehe auch [GKT90]). Darüber hinaus sind sämtliche überladenen Funktionen in den überladenen Argumentpositionen strikt, was einerseits eine Einschränkung für Sprachen mit verzögerter Auswertung bedeutet, andererseits aber mehr Informationen für die Striktheits-Analyse beinhaltet und damit die Generierung effizienten Codes erleichtert.[25]

Demgegenüber steht die Möglichkeit der Überladung von Konstanten, wenn man kontextabhängige Überladungen erlaubt, wie etwa die Überladung numerischer Literale in Haskell. Dies gestattet dann einen höheren Abstraktionsgrad beim Abfassen generischer Bibliotheksroutinen. So kann man beispielsweise eine generische Fibonacci-Funktion definieren, die für beliebige Zahltypen verwendbar ist oder etwa eine Funktion zum Summieren von Listen beliebiger Zahltypen. Diese Allgemeinheit ist jedoch nicht unproblematisch: durch die Übersetzung werden zusätzliche Parameter eingeführt und damit unterscheidet sich die operationale Semantik des übersetzten Programms erheblich von dem gleichen Programm mit vollständig aufgelösten Überladungen. So schließt beispielsweise [Aug93] mit dem Fazit, daß für eine effi-

[24] Natürlich können die Kennzeichen für vollständig aufgelöste Überladungen eingespart werden, was jedoch eine Datenflußanalyse im Übersetzer verlangt.

[25] SAMPλE verwendet eine vom Autor geschriebene Striktheitsanalyse auf Basis der Rückwärtsanalyse von John Hughes [Hug87].

ziente Implementierung kontextabhängiger Überladungen eine partielle Auswertung des generierten Programms notwendig ist, um eine mit dem nicht überladenen Fall gleichwertige Codequalität zu erreichen.

Die Möglichkeit der Gruppierung verwandter überladener Operatoren in Typklassen und die damit einhergehende Einführung eines neuen Namensraums hat den Vorteil, daß Haskell-Typausdrücke gegenüber SAMPλE-Typausdrücken etwas prägnanter sind, stellt aber keine signifikante Erweiterung dar. Das gleiche läßt sich für das Typsystem von SAMPλE erreichen, indem man für häufig auftretende Operatormengen entsprechende Namen einführt, etwa in der Form **operators** $Num = \{+, -, *, /\}$.[26] Dies vermeidet zusätzlich den Nachteil, die Operatoren in eine Hierarchie einordnen zu müssen.

Ein alternativer Überladungsansatz für rekursive Überladungen wurde von Duggan und Ophel vorgeschlagen [CDO93]: statt die Menge der möglichen Typen eines überladenen Operators über Überladungsschemata mit Instanzierungsregeln zu beschreiben, könnte man die Typmenge über einen regulären Sortenausdruck beschreiben, dessen kleinster Fixpunkt die Menge der zulässigen Argumenttypen beschreibt. Ein Sortenausdruck enthält neben den üblichen Typkonstruktoren noch einen Vereinigungsoperator $\sqcup$, sowie $\top$ als Platzhalter für die Sorte, die alle Typen umfaßt. Damit wird beispielsweise der Typ des Additionsoperators, der für ganze und Fließkommazahlen definiert sein soll, durch den Typausdruck $\forall\alpha : int \sqcup real \;.\; \alpha \to \alpha \to \alpha$ beschrieben. Die Identitätsfunktion hat in diesem System den Typ $\forall\alpha : \top \;.\; \alpha \to \alpha$, der Typ des SAMPλE-Gleichheitsoperators wird durch $\forall\alpha : \mu\kappa.int \sqcup real \sqcup bool \sqcup list(\kappa) \sqcup \kappa \times \kappa \sqcup ref(\top) \;.\; \alpha \to \alpha \to bool$ dargestellt.[27]

Die Autoren geben einen unitären Unifikationsalgorithmus für solcherart sortierte Typausdrücke an, der auf einem Algorithmus zur Berechnung des Schnittmengenoperators für Sortenausdrücke basiert. Im wesentlichen entspricht die Ausdruckskraft bzgl. der möglichen Überladungsinstanzen dieses Systems, dem System der parametrischen Überladungen. Die Autoren geben an, wie man Haskell-Typklassen-Deklarationen in

[26] In der aktuellen Implementierung müssen Typvariablen, die in Typannotationen im Programm verwendet werden, zusammen mit den zulässigen Operatoren als Typvariablenname deklariert werden.

[27] $\mu\kappa.\rho$ steht dabei für den regulären Baum, den man erhält, wenn man κ in ρ durch ρ ersetzt, d.h., die Lösung der rekursiven Gleichung $\kappa = \rho$. Siehe auch Kapitel 7.

entsprechende Sortenausdrücke umwandelt. Der Ansatz beinhaltet jedoch schwerwiegende Nachteile:

1. Typausdrücke überladener Operatoren sind wesentlich komplexer in der Notation, als äquivalente Typausdrücke in SAMPλE bzw. Haskell, wie man am obigen Typausdruck für polymorphe Gleichheit leicht sieht.

2. Da der Unifikationsalgorithmus die Leerheit von Sortenausdrücken überprüfen muß, erfordert er aufgrund des DEXPTIME-Resultats für die Überprüfung der Existenz monomorpher Grundtypinstanzen im schlimmsten Fall exponentiellen Zeitaufwand.

3. Das System ist nicht inkrementell und damit nicht zur Behandlung benutzerdefinierbarer Überladungen geeignet.

Der letzte Punkt ist ein absolutes Ausschlußkriterium für den Entwurf einer verwendbaren Programmiersprache auf Basis dieses Systems, daher wurde die Syntax der Sorten in späteren Arbeiten der Autoren um sogenannte offene Sorten erweitert [DO94, DCO96]. Offene Sorten sind dabei nichts anderes als Mengen von Operatornamen, wie sie im System der parametrischen Überladungen verwendet werden. Die Autoren geben eine Übersetzung ihrer Quellsprache in eine abstrakte Maschine an, die auf der Einführung von Typausdrücken als zusätzliche Parameter basiert, und zeigen, daß die Ausführung typisierbarer, übersetzter Programme typfehlerfrei ist.

Bedauerlicherweise beinhaltet die Beschränkung der Syntax von Überladungsschemata auf ein einziges Überladungssymbol, bzw. die Haskell-Beschränkung auf Typklassen mit einem einzigen Parameter, zusammen mit den Restriktionen der Syntax von Überladungsinstanzen überladener Operatoren, den Ausschluß einiger interessanter überladener Operationen. So lassen sich etwa polymorphe Selektionsoperationen auf n-Tupeln nicht behandeln. Hier hilft nur der Übergang zu Überladungsschemata mit mehreren Überladungssymbolen bzw. zu Typklassen mit mehreren Parametern.[28] Dies sah schon die erste Veröffentlichung zum Thema Typklassen vor [WB89], allerdings wurde dort die Frage der Typinferenz ausgeklammert.[29] Im folgenden Kapitel

[28] Die Erweiterung des Systems mit Konstruktorvariablen bzw. Konstruktorklassen reicht nicht aus.

[29] Die erste Veröffentlichung mit einem vollständigen Typinferenzalgorithmus für Typklassen war [NS91], in der ordnungssortierte Unifikation als Grundlage der Typinferenz gewählt wurde.

werden wir einen allgemeinen Rahmen vorstellen, der die systematische Untersuchung entsprechender Erweiterungen erlaubt.

Kapitel 4

Typinferenz mit Prädikaten

In diesem Abschnitt verallgemeinern wir das Damas-Milner System durch Hinzufügen einschränkender Bedingungen an die Inferenzregeln, d.h., wir erlauben in den Prämissen der Inferenzregeln Prädikate, welche die Menge der für eine Metavariable des Deduktionssystems instanzierbaren Typen charakterisieren. Wir zeigen, daß Typinferenz mit let-Polymorphismus in einem solchen System immer noch möglich ist, wenngleich für beliebige (totale) Prädikate die Typisierbarkeit nur semi-entscheidbar ist. In einem solchen System kann der inferierte allgemeinste Typ eines Ausdrucks u.U. leer sein.

Die Hauptmotivation, die zur Entwicklung dieses System geführt hat, war es, eine einheitliche Beschreibung von Typinferenzsystemen und Algorithmen für Konversionstypisierung und Überladungstypisierung zu erhalten. Dabei sollte insbesondere die Beschränkung auf eine einzige Überladungsposition für überladene Operatoren, wie im vorangegangenen Kapitel über Überladungstypisierung beschrieben, aufgehoben werden.

In einem monomorphen Typsystem ist die Einführung zusätzlicher Restriktionen in den Typisierungsregeln relativ unproblematisch. Man ergänzt einfach die zusätzlichen Bedingungen als Prämisse der Typregeln. So könnte man beispielsweise eine Regel für die Funktionsapplikation definieren, die eine Konversion des Typs der aktuellen Argumente in den Typ der formalen Parameter erlaubt.

$$\frac{\Gamma \vdash M_1 : t_f \rightarrow t_r \qquad \Gamma \vdash M_2 : t_a \qquad t_a \lhd t_f}{\Gamma \vdash M_1 M_2 : t_r}$$

In diesem Fall wird die Konversionsbedingung $t_a \lhd t_f$, für die natürlich eine formale Beschreibung vorliegen muß, nur während der Herleitung des Typs des Ausdrucks verwendet, existiert also ausschließlich auf der Meta-Ebene der Deduktion und nicht

auf der Objektebene: es ist unmöglich, und auch nicht notwendig, daß die verwendete Konversionsbedingung im Typ des Ausdrucks sichtbar wird.

Mit dem Übergang zu einem polymorphen System bricht dieser Ansatz jedoch zusammen: die Konvertierbarkeit polymorpher Typausdrücke, z.B. zweier Typvariablen $\alpha \triangleleft \beta$, ist schlicht nicht überprüfbar. Dieses Problem wird gelöst, indem man die Konversionsbedingungen als Teil der Typaussage aufführt. Typaussagen sind dann von der Form $C, \Gamma \vdash M : \tau$ und werden wie folgt gelesen: unter der Typannahme Γ, die jeder freien Variablen in M einen Typ zuordnet, hat M den Typ τ, und C ist die Menge der Bedingungen, die zusätzlich erfüllt sein müssen.[1]

Die Typsierungsregel der Applikation könnte damit wie folgt definiert werden:[2]

$$\frac{C, \Gamma \vdash M_1 : \tau_f \to \tau_r \qquad C, \Gamma \vdash M_2 : \tau_a \qquad C \Vdash \tau_a \triangleleft \tau_f}{C, \Gamma \vdash M_1 M_2 : \tau_r}$$

$C \Vdash D$ beschreibt dabei eine binäre Relation auf Restriktionsmengen und ist wie folgt zu lesen: D ist eine logische Konsequenz von C, d.h. falls eine Substitution S existiert, sodaß alle Restriktionen in C erfüllt sind, dann sind auch alle Restriktionen in D erfüllt.[3] Für unser Beispiel bedeutet das, daß die Konvertierbarkeit von τ_a in τ_f aus der Gesamtmenge der Restriktionen folgen muß, die zur Typisierung des Gesamtausdrucks benötigt werden. Dabei muß nicht unbedingt $\tau_a \triangleleft \tau_f \in C$ gelten.

Auch Überladungen lassen sich durch Restriktionen darstellen. Dazu definiert man für jeden Operator o ein Prädikat p_o, sodaß $p_o(\overline{t_n})$ wahr ist, wenn $\{\overline{\alpha_m} \mapsto \overline{t_m}\}\tau_o$ eine gültige Überladungsinstanz des Typs τ_o von p_o ist. Der polymorphe Typ des Operators wird dann durch $\forall\overline{\alpha_m}.\tau_o|C_o$ beschrieben, wobei $p_o(\overline{\beta_n}) \in C_o$ und $\overline{\beta_n} \subseteq \overline{\alpha_m}$ gilt.

Die Restriktionstypen für parametrisch überladene Operatoren können aus einer beliebigen Überladungsannahme auf systematische Weise konstruiert werden: Gegeben

[1] Alternativ könnte man natürlich auch schreiben $\Gamma \vdash M : \tau|C$, wie z.B. in [Smi94], aber die obige Schreibweise scheint natürlicher und wird daher auch in fast allen Veröffentlichungen zum Thema Konversionstypisierung verwendet.

[2] Und dies ist für SAMPλE tatsächlich der Fall.

[3] Der Begriff der logischen Konsequenz von Restriktionsmengen taucht erstmals in der Arbeit von J.C. Mitchell [Mit84] über Konversionstypisierung auf und wurde später von Fuh und Mishra in [FM88] detaillierter untersucht. Von dort stammt auch unsere Schreibweise.

$$
\begin{aligned}
= : &\quad \forall\alpha.\, \alpha \to \alpha \to \mathit{bool} \mid p_{=}(\alpha) \\
< : &\quad \forall\alpha.\, \alpha \to \alpha \to \mathit{bool} \mid p_{<}(\alpha) \\
+ : &\quad \forall\alpha.\, \alpha \to \alpha \to \alpha \mid p_{+}(\alpha) \\
- : &\quad \forall\alpha.\, \alpha \to \alpha \to \alpha \mid p_{-}(\alpha) \\
* : &\quad \forall\alpha.\, \alpha \to \alpha \to \alpha \mid p_{*}(\alpha)
\end{aligned}
$$

Abbildung 4.1: Überladene Operatoren für SAMPλE, in Restriktionsschreibweise

eine Überladungsannahme O, definiert man für jeden überladenen Operator x mit $O(x) = \langle \omega, s \rangle$ ein Prädikatsymbol p_x mit $\rho(x) = 1$ und den eingeschränkten Typ

$$p_o : \forall\alpha, \beta_n.\, \{\$ \mapsto \alpha\}\omega \mid \{p_o(\alpha)\} \cup \{p_y(\beta) \mid \beta_X \in \omega \wedge y \in X\}.$$

Als Beispiel sind einige der Restriktionstypen für vordefinierte SAMPλE-Operatoren sind in Tabelle 4.1 angegeben.

Der Vorteil der Restriktionsschreibweise gegenüber der Schreibweise mit ordnungssortierten Typvariablen, ist die Möglichkeit, auch überladene Operatoren spezifizieren zu können, die an mehreren Argumentpositionen und/oder der Resultatposition überladen sind. Beispielsweise könnte man eine überladene Addition spezifizieren, die auf beliebigen Kombinationen von Zahltypen definiert ist. Diese hätte dann den polymorphen Typ $\forall\alpha, \beta, \gamma.\alpha \to \beta \to \gamma | p_{+}(\alpha, \beta, \gamma)$, wobei $p_{+}(t_1, t_2, t_3)$ wahr wäre, wenn p_{+} bei Anwendung auf Werte vom Typ t_1 als erstem Argument und t_2 als zweitem Argument den Ergebnistyp t_3 liefert.

Die verbleibenden Abschnitte dieses Kapitels sind wie folgt organisert: zunächst (4.1) definieren wir die grundlegen Begriffe wie erfüllbare bzw. lösbare Restriktionsmengen, eingeschränkte Typen, Implikationsrelationen und Typinstanzrelationen formal und untersuchen ihre Eigenschaften. Anschließend (4.2) definieren wir ein generisches Typdeduktionssystem für eingeschränkte Typen, leiten einige Aussagen über die Typisierungsrelation her und präsentieren in Abschnitt 4.3 einen Typinferenzalgorithmus ($\mathcal{C}$) zur Berechnung allgemeinster Typen. Wir untersuchen den Begriff

der Lösungen und Vereinfachungen von Restriktionsmengen (4.4, 4.5), definieren auf Basis von Vereinfachungen einen verbesserten Typinferenzalgorithmus ($\mathcal{D}$) und beweisen dessen Korrektheit (4.6). Eine Diskussion der Ergebnisse und ein Vergleich mit anderen Arbeiten bildet den Abschluss des Kapitels.

4.1 Grundlegende Definitionen

Definition 4.1. Sei P eine Menge von Prädikatsymbolen mit Stelligkeit $\rho : P \to \mathbb{N}$. Die Menge der Prädikatterme über $T_F(\mathcal{V})$ ist definiert durch

$$\mathcal{P}(F, V) = \{p(\tau_1, \ldots, \tau_{\rho(p)}) \mid p \in P, \tau_i \in T_F(\mathcal{V})\}$$

Für einen Prädikatterm $r = p(\tau_1, \ldots, \tau_n)$ bezeichne $\mathit{args}(r)$ die Menge der Argumente $\{\tau_1, \ldots, \tau_n\}$, und $\mathit{cargs}(r)$ sei definiert durch $\mathit{cargs}(r) \stackrel{\text{def}}{=} \mathit{args}(r) - T_{F_0}(\mathcal{V})$. Anwendung von Substitutionen und Variablenbestimmung werden in der kanonischen Weise auf Prädikatterme erweitert:

$$S(p(\tau_1, \ldots, \tau_n)) = p(S(\tau_1), \ldots, S(\tau_n))$$
$$\mathcal{V}(p(\tau_1, \ldots, \tau_n)) = \mathcal{V}(\tau_1) \cup \cdots \cup \mathcal{V}(\tau_n)$$

Um überhaupt Aussagen über die Typisierbarkeit von Ausdrücken machen zu können, benötigen wir einen Erfüllbarkeitsbegriff für Mengen von Prädikattermen. Wir nehmen dazu an, daß für jedes Prädikatsymbol $p \in P$ eine totale Funktion $\hat{p} \in T_F^{\rho(p)} \to \mathbb{B}$ existiert. Eine Familie totaler Funktionen $(\hat{p})_{p \in P}$ mit $\hat{p} : T_F^{\rho(p)} \to \mathbb{B}$ nennen wir Interpretation.

Definition 4.2. Sei $C = \{p_i(\tau_{i1}, \ldots, \tau_{i\rho(p_i)}) \mid i \in I\}$ eine Menge von Prädikattermen. Eine Substitution $S \in V \to T_F$ *erfüllt* C ($S \models C$), falls für jeden Ausdruck $p(\tau_n) \in C$ die Interpretation $\hat{p}(S(\tau_1), \ldots, S(\tau_n))$ wahr ist. C heißt *erfüllbar*, falls eine erfüllende Substitution existiert, wir schreiben dann auch $\models C$ bzw. $\not\models C$, falls keine solche Substitution existiert.

Besteht die Prädikatmenge aus einem einzigen Prädikat, schreiben wir auch kurz p, statt $\{p\}$.

Proposition 4.3.

$$S \models \emptyset \tag{4.1}$$

$$S \models C \iff \forall p \in C.\, S \models p \tag{4.2}$$

$$S \circ R \models C \iff S \models R(C) \tag{4.3}$$

Beweis. (4.1) und (4.2) sind direkte Konsequenzen der Definition der Erfüllbarkeit. (4.3) folgt aus der Tatsache, daß $p(S(R(\overline{\tau_n}))) = S(p(R(\overline{\tau_n})))$. □

Definition 4.4. Eine Substitution Q heißt *Lösung* einer Prädikatmenge C, falls $L \models Q(C)$ für jede Substitution $L \in \mathcal{V} \to T_F$ mit $\mathcal{V}(Q(C)) \subseteq dom(L)$. Wir nennen eine Prädikattermmenge C *gelöst*, falls jede Substitution S die Menge C erfüllt.

Für ein System P von Prädikaten wollen wir die Menge der Lösungen einer Prädikatmenge C mit $\mathcal{L}_P(C)$ bezeichnen. Offensichtlich ist jede erfüllende Substitution einer Prädikattermmenge auch eine Lösung: $L \models C \Rightarrow L \in \mathcal{L}_P(C)$. Außerdem ist jede Substitution eine Lösung für eine gelöste Prädikattermmenge:

Lemma 4.5. $S \in \mathcal{L}_P(C) \Rightarrow \mathcal{L}_P(S(C)) = \mathcal{V} \to T_F(\mathcal{V})$

Beweis. Sei R eine Substitution und $Q \in \mathcal{V} \to T_F$ eine Substitution mit $dom(Q) \subseteq \mathcal{V}(R(S(C)))$. Offensichtlich gilt $Q \circ R \models S(C)$ und wg. Gleichung 4.3 auch $Q \models R(S(C))$, woraus $R \in \mathcal{L}_P(S(C))$ folgt. □

Definition 4.6 (Eingeschränkter Typ). Ein *eingeschränkter Typ* ist ein Paar $\tau|C$ wobei τ ein Typausdruck und C eine Menge von Prädikattermen ist. Entsprechend ist ein *eingeschränktes Typschema* ein Ausdruck der Form $\forall\overline{\alpha_n}.\tau|C$.

Eingeschränkte Typausdrücke stehen für eine Menge monomorpher Typen und zwar für all jene, die sich durch Substitution monomorpher Typen für Typvariablen ergeben, sodaß gleichzeitig alle Einschränkungen, d.h. Prädikate, erfüllt werden. So beschreibt etwa das Typschema $\forall\alpha, \beta.\alpha \to \beta|\alpha \lhd \beta$ all jene monomorphen Typen $t_1 \to t_2$, bei denen der Typ t_1 in den Typ t_2 konvertiert werden kann.

$$\mathcal{I}(\tau|C) = \{L(\tau) \mid \exists L.\, L \models C \wedge \mathcal{V}(\tau) \subseteq dom(L)\}$$

Essentiell für die Definition eines Instanzbegriffs zwischen eingeschränkten Typausdrücken, und damit auch eines Instanzbegriffs für Typisierungen mit Prädikatmengen, sowie für die Definition von Typdeduktionssystemen für eingeschränkte Typen, ist der Begriff der logischen Konsequenz (Implikation) von Prädikattermmengen ($C \Vdash D$). Die exakte Definition hängt von der Interpretation der betrachteten Prädikate ab, wir verlangen jedoch von einer sinnvollen Implikationsrelation einige sinnvolle Eigenschaften:

Definition 4.7. Eine Implikationsrelation $\Vdash$ auf Prädikattermmengen muß die folgenden Bedingungen erfüllen.[4]

Jede Lösung einer Prädikatmenge C, die eine Menge D impliziert, muß auch Lösung von D sein:

$$C \Vdash D \Rightarrow \mathcal{L}_P(C) \subseteq \mathcal{L}_P(D) \tag{4.4}$$

$\Vdash$ ist abgeschlossen unter Substitutionen:

$$C \Vdash D \Rightarrow S(C) \Vdash S(D) \tag{4.5}$$

$\Vdash$ ist transitiv und reflexiv:

$$C \Vdash D \wedge D \Vdash E \Rightarrow C \Vdash E \tag{4.6}$$

$$D \subseteq C \Rightarrow C \Vdash D \tag{4.7}$$

$\Vdash$ ist distributiv gegenüber Vereinigung von Prädikatmengen:[5]

$$C \Vdash D \wedge C \Vdash E \iff C \Vdash D \cup E \tag{4.8}$$

Gelöste Termmengen sind immer impliziert:

$$S \in \mathcal{L}_P(\, D\,) \Rightarrow C \Vdash S(D) \tag{4.9}$$

[4] Dabei hat $\Vdash$ eine syntaktische Priorität, die zwischen den logischen Operatoren und den Mengenoperatoren liegt. Also ist z.B. $C \Vdash D \wedge D \Vdash E \Rightarrow C \Vdash E$ äquivalent zu $(C \Vdash D) \wedge (D \Vdash E) \Rightarrow (C \Vdash E)$.

Lemma 4.8. *Falls die Implikationsrelation statt 4.4 die schärfere Bedingung*

$$C \Vdash D \iff \mathcal{L}_P(C) \subseteq \mathcal{L}_P(D) \tag{4.10}$$

erfüllt, folgen die in Definition 4.7 zusätzlich gestellten Bedingungen direkt aus 4.10.

Beweis. Wir zeigen nur (4.5) und (4.9): falls $S(C)$ nicht erfüllbar ist, d.h. $\not\models S(C)$, folgt die Behauptung aus $\mathcal{L}_P(S(C)) = \emptyset$. Falls $S(C)$ erfüllbar ist, existiert eine Substitution L, mit $L \models S(C)$. Dann folgt aber aus Gleichung 4.3 $L \circ S \models C$, und aus $C \Vdash D$ auch $L \circ S \models D$ und somit $L \models S(D)$, also gilt (4.5). (4.9) folgt aus Lemma 4.5, da $\mathcal{L}_P(S(D)) = \mathcal{V} \to T_F(\mathcal{V}) \supseteq \mathcal{L}_P(C)$, was wg. Gleichung (4.10) äquivalent ist zu $C \Vdash S(D)$. □

Man kann also sagen, daß in einem gewissen Sinn Bedingung 4.4 die wichtigste Eigenschaft einer Implikationsrelation ist, denn jede Implikationsrelation ist aufgrund von Lemma 4.8 eine Teilmenge der durch Gleichung 4.10 induzierten Relation. Allerdings hat Gleichung 4.10 zur Konsequenz, daß man Restriktionen beliebig austauschen kann, sofern sich die Lösungsmenge nicht ändert. Für den Fall, daß man die Semantik von Ausdrücken durch eine Übersetzung definiert, die von den konkreten Restriktionen abhängt, wie z.B. im Fall der parametrischen Überladungen, würden äquivalente Typisierungen zu semantisch unterschiedlichen Übersetzungen führen.[6] Daher läßt sich Gleichung i.A. nicht erfüllen.

Korrollar 4.9. $C \Vdash D \iff \forall p \in D.\ C \Vdash p$

Beweis. Folgt direkt aus Gleichung 4.8.[7] □

Korrollar 4.9 eröffnet eine elegante Möglichkeit zur Definition von Implikationsrelationen: ein Deduktionssystem für eine Relation $C \Vdash p$ zwischen Restriktionsmengen und einelementigen Prädikattermen kann zu einem Deduktionssystem für eine allgemeine Implikationsrelation erweitert werden, indem man die in Abbildung 4.2 angegebenen Regeln ergänzt.

[5] Manche Autoren schreiben $C \wedge D$ statt $C \cup D$.
[6] Man kann nicht ohne weiteres p_+ durch p_- ersetzen, ohne die Semantik zu ändern.
[7] Umgekehrt folgt Gleichung 4.8 auch aus $C \Vdash D \iff \forall p \in D.\ C \Vdash p$.

$$[\text{b1}]\quad C \Vdash \emptyset$$

$$[\text{b2}]\quad \frac{p \in C}{C \Vdash p}$$

$$[\text{b3}]\quad \frac{C \Vdash D \qquad D \Vdash E}{C \Vdash E}$$

$$[\text{b4}]\quad \frac{C \Vdash D \qquad C \Vdash E}{C \Vdash D \cup E}$$

Abbildung 4.2: Basisregeln für Implikationsdeduktionssysteme

Regeln (b0) und (b1) garantieren die Reflexivität (Gleichung 4.7), (b2) die Transitivität (Gleichung 4.6) und Regel (b3) die Distributivität (Gleichung 4.8) für jede durch Erweiterung definierte Implikationsrelation. Darüber hinaus sind (4.4) und (4.5) erfüllt, sodaß es genügt, die notwendigen Bedingungen einer Implikationsrelation für die zusätzlichen Regeln zu überprüfen.

Als Beispiel betrachten wir die Definition einer Implikationsrelation zwischen Prädikattermen mit dem Symbol $=$ für syntaktische Gleichheit, deren Regeln in Abbildung 4.3 angegeben sind. Die Regeln e1, e2 und e3 spezifizieren Reflexivität, Transitivität und Kommutativität der syntaktischen Gleichheit. Regel e4 definiert, daß die Gleichheit zusammengesetzter Terme aus der Gleichheit der Subterme folgt. Regel e5 legt fest, daß aus der ableitbaren Gleichheit zweier zusammengesetzter Typausdrücke mit identischen Konstruktoren auch die Gleichheit der Subterme an identischen Termpositionen folgt.[8]

Lemma 4.10. *Die durch die Regeln b1-b4 und e1-e5 definierte Relation ist eine Implikationsrelation.*

[8] Auf den ersten Blick scheint diese Regel etwas unnatürlich zu sein. Ihre Notwendigkeit kann man sich an folgendem Beispiel vor Augen führen: Sei $C = \{\alpha = \beta\}$ und $D = \{f(\alpha) = f(\beta)\}$. Offensichtlich spezifizieren beide Restriktionsmengen identische Einschränkungen an die Typvariablen α und β. Ohne Regel e5 gilt aber nur $C \Vdash D$ und nicht $D \Vdash C$.

$$[\text{e1}]\quad C \Vdash \tau = \tau$$

$$[\text{e2}]\quad \frac{C \Vdash \tau_1 = \tau_2 \qquad C \Vdash \tau_2 = \tau_3}{C \Vdash \tau_1 = \tau_3}$$

$$[\text{e3}]\quad \frac{C \Vdash \tau_1 = \tau_2}{C \Vdash \tau_2 = \tau_1}$$

$$[\text{e4}]\quad \frac{C \Vdash \tau_1 = \tau_1', \ldots, \tau_n = \tau_n' \qquad f \in F_n}{C \Vdash f(\overline{\tau_n}) = f(\overline{\tau_n'})}$$

$$[\text{e5}]\quad \frac{C \Vdash f(\overline{\tau_n}) = f(\overline{\tau_n'})}{C \Vdash \tau_1 = \tau_2, \ldots, \tau_n = \tau_n'}$$

Abbildung 4.3: Implikationsregeln für syntaktische Gleichheit

Beweis. Da das Prädikatsymbol $=$ für syntaktische Gleichheit steht, wissen wir, daß jede Lösung μ ein Unifikator der in C enthaltenen Gleichungen ist. Somit sind alle Gleichungen in $\mu(C)$ von der Form $\tau = \tau$ und daher garantiert e1 die Bedingung (4.9). Die Abgeschlossenheit unter Substitutionen folgt durch Induktion über die Länge der Deduktion einer Aussage $C \Vdash D$. Somit verbleibt zu zeigen, daß Gleichung (4.4) erfüllt ist. Durch Induktion über die Länge einer Deduktion zeigen wir die Implikation $C \Vdash D \Rightarrow \mathcal{L}_P(C) \subseteq \mathcal{L}_P(D)$: Für e1 folgt die Aussage aus Lemma 4.5. Für e2 folgt aus der Induktionshypothese, daß jede Lösung von C auch eine Lösung für $\tau_1 = \tau_2$ und $\tau_2 = \tau_3$ ist, d.h. $\mu(\tau_1) = \mu(\tau_2)$ und $\mu(\tau_2) = \mu(\tau_3)$ und somit aufgrund der Transitivität der syntaktischen Gleichheit auch $\mu(\tau_1) = \mu(\tau_3)$. Für e3 bis e5 haben die betrachteten Aussagen identische Lösungsmengen. □

Man beachte, daß die so definierte Implikationsrelation Gleichung (4.10) nicht erfüllt: wähle $C = \{a = b\}$ und $D = \{c = d\}$ mit $a \neq b$ und $c \neq d$ und $a, b, c, d \in F_0$. Offensichtlich sind C und D unlösbar und damit gilt natürlich $\mathcal{L}(C) = \mathcal{L}(D) = \emptyset$. Andererseits läßt sich aber keine der Deduktionsregeln anwenden, um $C \Vdash D$ abzuleiten. Fügt man jedoch die Regel $\not\models C \Rightarrow C \Vdash p$ zu dem Regelsystem hinzu,

$$[\text{k1}]\quad \frac{\hat{\lhd}(a,b) \qquad a,b \in F_0}{C \Vdash a \lhd b}$$

$$[\text{k2}]\quad C \Vdash \tau \lhd \tau$$

$$[\text{k3}]\quad \frac{C \Vdash \tau_1 \lhd \tau_2 \qquad C \Vdash \tau_2 \lhd \tau_3}{C \Vdash \tau_1 \lhd \tau_3}$$

$$[\text{c1}]\quad \frac{C \Vdash \tau_a' \lhd \tau_a \qquad C \Vdash \tau_r \lhd \tau_r'}{C \Vdash \tau_a \to \tau_r \lhd \tau_a' \to \tau_r'}$$

$$[\text{c2}][9]\quad \frac{C \Vdash \tau_a \to \tau_r \lhd \tau_a' \to \tau_r'}{C \Vdash \tau_a' \lhd \tau_a \qquad C \Vdash \tau_r \lhd \tau_r'}$$

Abbildung 4.4: Implikationsregeln für strukturelle Konversionen

$$[\text{eq1}]\quad \frac{\widehat{eq}(a) \qquad a \in F_0}{C \Vdash eq(a)}$$

$$[\text{eq2}]\quad \frac{C \Vdash eq(\tau_1) \qquad C \Vdash eq(\tau_2)}{C \Vdash eq(\tau_1 \times \tau_2)}$$

$$[\text{eq3}]\quad \frac{C \Vdash eq(\tau_1 \times \tau_2)}{C \Vdash eq(\tau_1) \qquad C \Vdash eq(\tau_2)}$$

Abbildung 4.5: Implikationsregeln für überladene Gleichheit

wird $\Vdash$ vermutlich vollständig, d.h. $\mathcal{L}_P(C) \subseteq \mathcal{L}_P(D) \Rightarrow C \Vdash D$.

Enthält die Menge der Prädikatterme zusätzlich zum Gleichheitssymbol weitere Prädikate, benötigt man noch die Kontextregel

$$[\text{c}]\quad \frac{C \Vdash p[\tau_1] \qquad C \Vdash \tau_1 = \tau_2}{C \Vdash p[\tau_2]}$$

Dabei bezeichnet $p[\tau]$ einen Prädikatterm, der τ als Teilterm enthält.

Abbildung 4.4 enthält ein Deduktionssystem für strukturelle Konversionen.

Wir nennen zwei Prädikatmengen äquivalent ($C \equiv D$), falls $C \Vdash D$ und $D \Vdash C$ gilt. Man sieht leicht, daß $\equiv$ eine Äquivalenzrelation ist und somit $\Vdash$ eine partielle Ordnung auf den gemäß $\equiv$ faktorisierten Prädikattermmengen bildet.[10]

Die durch $\Vdash$ implizierte Äquivalenzrelation bietet die Möglichkeit, gegebene Typaussagen $C, \Gamma \vdash M : \tau$ zu vereinfachen, indem man C durch eine „einfachere" Restriktionsmenge D mit $C \equiv D$ ersetzt, die u.U. weniger Restriktionen erhält. Beispielsweise gilt für die Typisierung struktureller Konversionen[11], daß die Mengen $C_1 = \{\alpha \times \beta \lhd \gamma \times \delta, \alpha \lhd \gamma, \beta \lhd \delta\}$ und $C_2 = \{\alpha \lhd \gamma, \beta \lhd \delta\}$ äquivalente Einschränkungen spezifizieren, wobei C_2 sicherlich einfacher ist, da $C_2 \subseteq C_1$. Andererseits gilt für $C_3 = \{\alpha \times \beta \lhd \gamma \times \delta\}$ aber auch $C_3 \equiv C_1$, und C_3 scheint eine mit C_2 vergleichbare Komplexität zu besitzen, da die Anzahl der Symbole gleich ist. Somit ist klar, daß zur Definition einer Vereinfachungsstrategie zusätzliche Festlegungen zur Messung der Komplexität getroffen werden müssen.

Die folgenden Rechenregeln können zur Vereinfachung von Restriktionsmengen angewendet werden:

Proposition 4.11.

$$C \Vdash D \Rightarrow C \equiv C \cup D \tag{4.11}$$

$$L \in \mathcal{L}_P(C) \Rightarrow L(C) \equiv \emptyset \tag{4.12}$$

$$L \models C \Rightarrow L(C) \equiv \emptyset \tag{4.13}$$

[9] Für jeden Typkonstruktor werden Regeln analog c1 und c2 benötigt. Also beispielsweise

$$\text{[c3]} \quad \frac{C \Vdash \tau_1 \lhd \tau_1' \qquad C \Vdash \tau_2 \lhd \tau_2'}{C \Vdash \tau_1 \times \tau_2 \lhd \tau_1' \times \tau_2'}$$

$$\text{[c4]} \quad \frac{C \Vdash \tau_1 \times \tau_2 \lhd \tau_1' \times \tau_2'}{C \Vdash \tau_1 \lhd \tau_1' \qquad C \Vdash \tau_2 \lhd \tau_2'}$$

Falls die Konversionsrelation eine partielle Ordnung ist, fügt man außerdem noch die folgende Regel hinzu:

$$\text{[k4]} \quad \frac{C \Vdash \tau_1 \lhd \tau_2 \qquad C \Vdash \tau_2 \lhd \tau_1}{C \Vdash \tau_1 = \tau_2}$$

[10] Die Reflexivität folgt aus Gleichung 4.7.

[11] Siehe Kapitel 6.

Beweis. (4.11) Trivialerweise gilt $C \cup D \Vdash C$. Umgekehrt folgt aus $C \Vdash C$ und $C \Vdash D$ aber auch $C \Vdash C \cup D$. Gleichung 4.12 folgt aus Gleichung 4.9. Gleichung 4.13 folgt aus Gleichung 4.12, da jede erfüllende Substitution auch eine Lösung ist. □

Gleichung 4.11 erlaubt es, Prädikatterme aus einer Prädikattermmenge zu entfernen, die von der verbleibenden Menge impliziert werden, ohne daß sich die Lösungsmenge ändert. Formal: $C \Vdash p \Rightarrow C \cup \{p\} \equiv C$. Gleichung (4.12) impliziert, daß alle gelösten Prädikattermmengen äquivalent zur leeren Menge sind.

Die Definition der Begriffe Menge der freien bzw. gebundenen Variablen wird in kanonischer Weise von einfachen Typausdrücken auf eingeschränkte Typen bzw. Typschemata fortgesetzt:

$$\mathcal{V}(\tau|C) = \mathcal{V}(\tau) \cup \mathcal{V}(C)$$
$$\mathcal{FV}(\overline{\alpha_n}.\tau|C) = \mathcal{V}(\tau|C) - \{\alpha_1, \ldots, \alpha_n\}$$

Definition 4.12 (Instanzbegriff für eingeschränkte Typschemata). Sei $S = \{\alpha_i \mapsto \tau_i\}$ eine Substitution. Ein eingeschränkter Typ $\tau'|C'$ ist eine *Instanz* von $\tau|C$, geschrieben $\tau'|C' \leq \tau|C$, falls $\tau' = S(\tau)$ und $C' \Vdash S(C)$. Für ein eingeschränktes Typschema $\sigma = \forall\overline{\alpha_n}.\tau|C$ bezeichnet $S(\sigma)$ das Typschema σ', das man durch Instanzierung der freien Vorkommen der α_i in τ durch τ_i erhält. Dabei müssen u.U. die in σ gebundenen Variablen umbenannt werden, um Namenskollisionen zu vermeiden. Ein Typschema $\sigma' = \forall\overline{\beta_m}.\tau'|C'$ heißt *generische Instanz* eines Typschemas $\sigma = \forall\overline{\alpha_n}.\tau|C$, geschrieben $\sigma' \prec \sigma$, falls $C' \Vdash S(C)$, $\tau' = S(\tau)$ und keines der β_i frei in τ vorkommt.

Proposition 4.13. *Die Instanzrelationen für eingeschränkte Typen bzw. Typschemata sind reflexiv, transitiv und abgeschlossen unter Substitutionen.*[12]

Beweis. Wir zeigen nur die Abgeschlossenheit unter Substitutionen, Reflexivität und Transitivität folgen analog: Es gelte $\sigma_1 \prec \sigma_2$ mit $\sigma_1 = \forall\overline{\alpha_n}.\tau_1|C_1$ und $\sigma_2 = \forall\overline{\beta_n}.\tau_2|C_2$. O.B.d.A. nehmen wir an, daß $inv(S) \cap (\mathcal{BV}(\sigma_1) \cup \mathcal{BV}(\sigma_2)) = \emptyset$.[13] Nach Definition der generischen Instanz existiert eine Substitution R, sodaß $\tau_1 = R\tau_2$

[12] Siehe Proposition 2.17 auf Seite 20.
[13] Dies kann durch α-Konversion immer erreicht werden.

und $C_1 \Vdash R(C_2)$. Offensichtlich gilt $S(\tau_1) = S(R(\tau_2))$ und da Implikationsrelationen abgeschlossen unter Substitution sind, gilt auch $S(C_1) \Vdash S(R(C_2))$. Somit folgt $S(\sigma_1) \prec S(\sigma_2)$. □

Man beachte, daß Proposition 2.17(ii) für eingeschränkte Typschemata i.A. nicht gilt: $\sigma_1 \prec \sigma_2 \not\Rightarrow \mathcal{FV}(\sigma_1) \supseteq \mathcal{FV}(\sigma_2)$. Gegenbeispiel: $\sigma_1 = int|\emptyset$, $\sigma_2 = \forall\alpha.\alpha|\beta = \beta$. Mit $S = \{\alpha \mapsto int\}$ gilt $int = S(\alpha)$ und $\emptyset \Vdash \{\beta = \beta\}$, aber nicht $\emptyset \supseteq \{\beta\}$.

Falls $\tau_1|C_1 \leq \tau_2|C_2$ und $\tau_2|C_2 \leq \tau_1|C_1$ so schreiben wir $\tau_1|C_1 \equiv \tau_2|C_2$. Offensichtlich ist $\leq$ eine Äquivalenzrelation. Desgleichen schreiben wir $\sigma_1 \equiv_\prec \sigma_2$, falls $\sigma_1 \prec \sigma_2$ und $\sigma_2 \prec \sigma_1$. Desweiteren $\Gamma_1 \equiv_\prec \Gamma_2$ falls $\Gamma_1(x) \equiv_\prec \Gamma_2(x)$ für alle $x \in dom(\Gamma_2)$.

Definition 4.14 (Generalisierung eingeschränkter Typen). $\mathrm{gen}(\Gamma, \tau|C)$ bezeichnet die Generalisierung eines eingeschränkten Typs im Kontext der Typannahme Γ; es ist das Typschema $\forall\overline{\alpha_n}.\tau|C'$, sodaß $\{\alpha_1, \ldots, \alpha_n\} = \mathcal{V}(\tau|C) - \mathcal{FV}(\Gamma)$ und $C' = \{p(\overline{\tau_m}) \in C \mid \mathcal{V}(p(\overline{\tau_m})) \cap \{\overline{\alpha_n}\} \neq \emptyset\}$.

Die folgenden Eigenschaften der Generalisierung werden für die Beweise des nächsten Abschnitts benötigt:

Proposition 4.15 (Eigenschaften von $\mathrm{gen}(\Gamma, \tau|C)$**).** [14]

$$\Gamma' \prec \Gamma \wedge \mathcal{FV}(\Gamma') \supseteq \mathcal{FV}(\Gamma) \Rightarrow \mathrm{gen}(\Gamma', \tau|C) \prec \mathrm{gen}(\Gamma, \tau|C) \wedge \mathcal{FV}(\mathrm{gen}(\Gamma', \tau|C)) \supseteq \mathcal{FV}(\mathrm{gen}(\Gamma, \tau|C)) \quad \text{(i)}$$

$$\mathrm{gen}(S\Gamma, S(\tau|C)) \prec S(\mathrm{gen}(\Gamma, \tau|C)) \quad \text{(ii)}$$

$$inv(S) \cap (\mathcal{V}(\tau|C) - \mathcal{FV}(\Gamma)) = \emptyset \Rightarrow \mathrm{gen}(S\Gamma, S(\tau|C))) = S(\mathrm{gen}(\Gamma, \tau|C)) \quad \text{(iii)}$$

$$R \text{ Variablenumbenennung auf } \mathcal{V}(\tau|C) \wedge rng(R) \cap \mathcal{FV}(\Gamma) = \emptyset \Rightarrow \mathrm{gen}(R\Gamma, R(\tau|C))) = R(\mathrm{gen}(\Gamma, \tau|C)) \quad \text{(iv)}$$

Beweis. (i): Aus der Annahme $\mathcal{FV}(\Gamma') \supseteq \mathcal{FV}(\Gamma)$ folgt $\mathrm{gen}(\Gamma', \tau|C) = \forall\overline{\alpha_n}.\tau|C = \sigma'$ und $\mathrm{gen}(\Gamma, \tau|C) = \forall\overline{\beta_m}.\tau|C = \sigma$, wobei $\overline{\alpha_n} \supseteq \overline{\beta_m}$ und damit auch $\mathcal{FV}(\sigma') \supseteq \mathcal{FV}(\sigma)$.

Für (ii) sei $\sigma' = \forall\overline{\beta_m}.S(\tau|C) = \mathrm{gen}(S\Gamma, S(\tau|C))$ und $\sigma = \forall\overline{\alpha_n}.\tau|C = \mathrm{gen}(\Gamma, \tau|C)$. Dann existiert eine Variablenumbenennung R mit $dom(R) = \{\overline{\alpha_n}\}$ und $rng(R) \cap inv(S) = \emptyset$, sodaß $\sigma = \forall R(\overline{\alpha_n}).R(\tau|C)$ und damit $S\sigma = \forall R(\overline{\alpha_n}).S(R(\tau|C))$. Zu

[14] Siehe Proposition 2.19 auf Seite 22.

zeigen ist, daß eine Substitution Q existiert, sodaß $S\tau = QSR\tau$, $SC \Vdash QSRC$ und $dom(Q) = rng(R)$. Wähle $Q(R\alpha) = S\alpha$. Es gilt $\forall\alpha \in \mathcal{V}(\tau) : S\alpha = Q(S(R\alpha))$, denn: falls $\alpha \in \mathcal{FV}(\Gamma)$ gilt $\alpha \notin inv(R)$ und daher $Q(S(R\alpha)) = Q(S\alpha)$. Da $dom(Q) = rng(R)$ und $rng(R) \cap inv(S) = \emptyset$, folgt $Q(S\alpha) = S\alpha$. Für $\alpha \notin \mathcal{FV}(\Gamma)$ gilt $\alpha \in dom(R)$ und $R\alpha \notin dom(S)$. Dies impliziert $Q(S(R\alpha)) = Q(R\alpha)$, was nach Konstruktion von Q aber $S\alpha$ ist. Damit folgt $S\tau = QSR\tau$ und $SC = QSRC$.

(iii) folgt wegen $S(\forall\overline{\alpha_n}.\tau|C) = \forall\overline{\alpha_n}.\tau|C$ sofort und (iv) ist offensichtlich. □

4.2 Generische Typdeduktion

Prinzipiell ist eine Vielzahl von Deduktionssystemen für eingeschränkte Typen denkbar, die sich nur in der Wahl der Einschränkungen unterscheiden, die in den einzelnen Regeln verwendet werden. Um nicht für jedes dieser Typinferenzsysteme einen eigenen Typinferenzalgorithmus angeben zu müssen, definieren wir nun ein generisches Inferenzsystem, mit dem sich alle für uns im Rahmen dieser Arbeit interessanten Deduktionssysteme darstellen lassen. Die grundlegende Idee ist dabei die Parametrisierung der Deduktionsregeln durch die Einführung dreier Funktionen $R_{var}(\tau)$, $R_{abs}(\tau)$ und $R_{app}(\tau_1, \tau_2)$, die mit den Typen parametrisiert sind, die in den Prämissen der Regeln vorkommen. Dabei dürfen durch Anwendung der Funktionen keine ungebunden Typvariablen eingeführt werden und abgeschlossen unter Substitutionen sein:

$$\mathcal{FV}(R_x(\overline{\tau_n})) = \mathcal{V}(\overline{\tau_n}) \tag{4.14}$$

$$R_x(S(\overline{\tau_n})) = S(R_x(\overline{\tau_n})). \tag{4.15}$$

Die einzige unparametrisierte Regel, und somit in allen Systemen identisch, ist die LET-Regel. Die LET-Regel beschreibt, daß der Ausdruck **let** $x = M_1$ **in** M_2 mit dem eingeschränkten Typ $\tau_2|C$ typisiert werden kann, vorausgesetzt $\tau_1|C_1$ ist ein eingeschränkter Typ für M_1 im Kontext Γ, $\tau_2|C$ ist ein eingeschränkter Typ für M_2 in dem um die Typannahme $x : \sigma$ erweiterten Kontext, wobei σ die Verallgemeinerung von $\tau_1|C_1$ im Kontext Γ darstellt. Die Bedingung $C \Vdash C_1$ verhindert, daß M_1 im Fall $x \notin \mathcal{FV}(M_2)$ untypisierbar sein darf.

Die LET-Regel kann wiederum äquivalent ersetzt werden durch eine Regel, die ohne

$$[\text{VAR}]\ \frac{\tau_0|C_0 \prec \Gamma(x) \qquad \tau_1|C_1 \prec R_{var}(\tau_0) \qquad C \Vdash C_0 \cup C_1}{C, \Gamma \vdash x : \tau_1}$$

$$[\text{ABS}]\ \frac{C, \Gamma + [x : \tau] \vdash M : \tau' \qquad \tau''|C_1 \prec R_{abs}(\tau, \tau') \qquad C \Vdash C_1}{C, \Gamma \vdash \lambda x.M : \tau''}$$

$$[\text{APP}]\ \frac{C, \Gamma \vdash M_1 : \tau' \qquad C, \Gamma \vdash M_2 : \tau \qquad \tau''|C_1 \prec R_{app}(\tau, \tau') \qquad C \Vdash C_1}{C, \Gamma \vdash M_1 M_2 : \tau''}$$

$$[\text{LET}]\ \frac{C_1, \Gamma \vdash M_1 : \tau \qquad C, \Gamma + [x : \text{gen}(\Gamma, \tau|C_1)] \vdash M_2 : \tau' \qquad C \Vdash C_1}{C, \Gamma \vdash \textbf{let}\ x = M_1\ \textbf{in}\ M_2 : \tau'}$$

Abbildung 4.6: Ein generisches Deduktionssystem mit eingeschränkten Typen

Generalisierung auskommt und sich stattdessen auf die Substitution des Unterausdrucks M_1 für jedes Vorkommen von x in M_2 stützt:

$$[\text{LET}]\ \frac{C_1, \Gamma \vdash M_1 : \tau_1 \qquad C, \Gamma \vdash M_2[M_1/x] : \tau_2 \qquad C \Vdash C_1}{C, \Gamma \vdash \textbf{let}\ x = M_1\ \textbf{in}\ M_2 : \tau}$$

Auch in dieser Formulierung ist entscheidend, daß C_1 durch C impliziert wird, sonst wären untypisierbare Unterausdrücke zulässig.

Das Milner System ist eine spezielle Instanz eines Inferenzsystems mit der Gleichheit als einzigem Prädikatsymbol.

$$R_{var}(\tau_i) = \tau_i|\emptyset$$
$$R_{abs}(\tau_a, \tau_r) = \tau_a \to \tau_r|\emptyset$$
$$R_{app}(\tau_f, \tau_a) = \forall\alpha.\ \alpha|\{\tau_f{=}\tau_a \to \alpha\}$$

Nach Vereinfachung der Deduktionsregeln, die durch direktes Einsetzen in das generische System entstehen, erhält man das in Abbildung 4.7 angegebene System. Ein Typsystem, das Konversionen von Funktionsargumenten erlaubt, wird durch

$$R_{var}(\tau_i) = \tau_i|\emptyset$$

$$R_{abs}(\tau_a, \tau_r) = \tau_a \to \tau_r | \emptyset$$
$$R_{app}(\tau_f, \tau_a) = \forall \alpha, \beta.\ \beta | \{\tau_f = \alpha \to \beta,\ \tau_a \lhd \alpha\}$$

definiert[15], während

$$R_{var}(\tau_i) = \forall \alpha.\ \alpha | \{\tau_i \lhd \alpha\}$$
$$R_{abs}(\tau_a, \tau_r) = \tau_a \to \tau_r | \emptyset$$
$$R_{app}(\tau_f, \tau_a) = \forall \alpha.\ \alpha | \{\tau_f = \tau_a \to \alpha\}$$

ein Typsystem beschreibt, in dem nur Variablen konvertiert werden. Das durch

$$R_{var}(\tau_i) = \forall \alpha.\ \alpha | \{\tau_i \lhd \alpha\}$$
$$R_{abs}(\tau_a, \tau_r) = \forall \alpha.\ \alpha | \{\tau_a \to \tau_r \lhd \alpha\}$$
$$R_{app}(\tau_f, \tau_a) = \forall \alpha, \beta.\ \beta | \{\tau_f = \tau_a \to \beta,\ \beta \lhd \alpha\}$$

implizit definierte Typsystem erlaubt dagegen Konversionen an jeder Stelle der Deduktion.

Lemma 4.16. *Sei $C, \Gamma \vdash M : \tau$ eine ableitbare Typaussage mit erfüllbarer Restriktionsmenge C. Dann sind alle in der Ableitung verwendeten Restriktionsmengen erfüllbar.*

Beweis. Folgt aus der Tatsache, daß alle in den Prämissen verwendeten Restriktionsmengen D entweder identisch zu C sind, oder zumindest in der Implikationsrelation stehen ($C \Vdash D$). □

Proposition 4.17. *Sei $C, \Gamma \vdash M : \tau$ eine ableitbare Typaussage und es gelte $D \Vdash C$. Dann ist auch $D, \Gamma \vdash M : \tau$ ableitbar.*

Beweis. Folgt durch Induktion über M aus der Transitivität von $\Vdash$. □

Somit folgt aus $C \equiv D$ und $C, \Gamma \vdash M : \tau$ auch $D, \Gamma \vdash M : \tau$. Außerdem folgt wegen $C \cup D \Vdash C$, daß bei gegebener ableitbarer Typaussage $C, \Gamma \vdash M : \tau$ durch Hinzunahme zusätzlicher Restriktionen eine neue ableitbare Typaussage $C \cup D, \Gamma \vdash M : \tau$ entsteht.

[15] Nach Vereinfachung erhält man das in Abbildung 4.8 angegebene Typdeduktionssystem.

$$[\text{VAR}]\ \frac{\tau | D \prec \Gamma(x) \qquad C \Vdash D}{C, \Gamma \vdash x : \tau}$$

$$[\text{ABS}]\ \frac{C, \Gamma + [x : \tau] \vdash M : \tau'}{C, \Gamma \vdash \lambda x.M : \tau \to \tau'}$$

$$[\text{APP}]\ \frac{C, \Gamma \vdash M_1 : \tau \qquad C, \Gamma \vdash M_2 : \tau' \qquad C \Vdash \{\tau = \tau' \to \tau''\}}{C, \Gamma \vdash M_1 M_2 : \tau''}$$

$$[\text{LET}]\ \frac{C_1, \Gamma \vdash M_1 : \tau \qquad C, \Gamma + [x : \text{gen}(\Gamma, \tau | C_1)] \vdash M_2 : \tau' \qquad C \Vdash C_1}{C, \Gamma \vdash \textbf{let}\ x = M_1\ \textbf{in}\ M_2 : \tau'}$$

Abbildung 4.7: Deduktionssystem für parametrischen Polymorphismus (als Prädikatsystem)

Proposition 4.18. *Sei $C, \Gamma' \vdash M : \tau$ eine ableitbare Typaussage und es gelte $\Gamma' \prec \Gamma$, sowie $\mathcal{FV}(\Gamma') \supseteq \mathcal{FV}(\Gamma)$. Dann ist auch $C, \Gamma \vdash M : \tau$ eine ableitbare Typaussage.*

Beweis. Durch Induktion über M. Die Gültigkeit der VAR-Regel folgt aus der Transitivität der Instanzrelation eingeschränkter Typausdrücke in der VAR-Regel. Die Regeln ABS und APP folgen direkt aus der Induktionshypothese. Für die LET-Regel benötigt man Proposition 4.15(i). □

Korrollar 4.19. *Ist $C, \Gamma \vdash M : \tau$ ableitbar, und es gelte $\Gamma \equiv_\prec \Gamma'$ mit $\mathcal{FV}(\Gamma') \supseteq \mathcal{FV}(\Gamma)$, dann ist auch $C, \Gamma' \vdash M : \tau$ ableitbar.*

Beweis. Folgt direkt aus Proposition 4.18. □

Proposition 4.20. *Sei $C, \Gamma \vdash M : \tau$ eine ableitbare Typaussage. Dann ist auch $S(C), S(\Gamma) \vdash M : S(\tau)$ ableitbar.*

Beweis. Folgt durch Induktion über M aus der Abgeschlossenheit von $\Vdash$ bzgl. Anwendung von Substitutionen.

$$[\text{VAR}]\ \frac{\tau|D \prec \Gamma(x) \qquad C \Vdash D}{C, \Gamma \vdash x : \tau}$$

$$[\text{ABS}]\ \frac{C, \Gamma + [x : \tau] \vdash M : \tau'}{C, \Gamma \vdash \lambda x.M : \tau \to \tau'}$$

$$[\text{APP}]\ \frac{C, \Gamma \vdash M_1 : \tau \qquad C, \Gamma \vdash M_2 : \tau_a \qquad C \Vdash \{\tau = \tau_f \to \tau_r, \tau_a \lhd \tau_f\}}{C, \Gamma \vdash M_1 M_2 : \tau_r}$$

$$[\text{LET}]\ \frac{C_1, \Gamma \vdash M_1 : \tau \qquad C, \Gamma + [x : \text{gen}(\Gamma, \tau|C_1)] \vdash M_2 : \tau' \qquad C \Vdash C_1}{C, \Gamma \vdash \textbf{let}\ x = M_1\ \textbf{in}\ M_2 : \tau'}$$

Abbildung 4.8: Deduktionssystem für parametrischen Polymorphismus mit Konversion von Funktionsargumenten (als Prädikatsystem)

Für $M \equiv x \in V$ gilt $C, \Gamma \vdash M : \tau$ genau dann, wenn Typen $\tau_0|C_0$ und $\tau_1|C_1$ existieren, sodaß $\tau_0|C_0 \prec \Gamma(x)$, $\tau|C_1 \prec R_{var}(\tau_0)$ und $C \Vdash C_0 \cup C_1$. Aus der Abgeschlossenheit generischer Instanz und Implikationsrelationen unter Substitutionen (Proposition 4.13) folgt aber $S\tau_0|SC_0 \prec S\Gamma(x))$, $S\tau_1|SC_1 \prec S(R_{var}(\tau_0))$ und $SC \Vdash SC_0 \cup SC_1$, und wg. $S(\Gamma(x) = (S\Gamma)(x)$ sowie $S(R_{var}(\tau_0)) = R_{var}(S(\tau_0))$ (Gleichung 4.14) ist $SC, S\Gamma \vdash M : S\tau$ eine gültige Typisierung. Für Abstraktionen und Applikationen folgt die Aussage analog aus der Induktionshypothese. Sei also $C, \Gamma \vdash \textbf{let}\ x = M_1\ \textbf{in}\ M_2 : \tau$ eine gültige Typisierung. Dann existieren Typisierungen $C_1, \Gamma \vdash M_1 : \tau_1$ und $C, \Gamma + [x : \text{gen}(\Gamma, \tau_1|C_1)] \vdash M_2 : \tau$ und $C \Vdash C_1$. Sei R eine Variablenumbenennung mit $dom(R) = \mathcal{V}(\tau_1|C_1) - \mathcal{FV}(\Gamma)$ und $cod(R) \cap inv(S) = \emptyset$. Aufgrund der Induktionshypothese ist $S(RC_1), S(R\Gamma) \vdash M_1 : S(R\tau_1)) = S(RC_1), S\Gamma \vdash M_1 : S(R\tau_1)$ eine gültige Typisierung und es gilt $inv(S) \cap (\mathcal{V}(R(\tau_1|C_1)) - \mathcal{FV}(\Gamma)) = \emptyset$. Damit folgt aus Proposition 4.15(iii): $\text{gen}(S\Gamma, S(R(\tau_1|C_1))) = S(\text{gen}(\Gamma, R(\tau_1|C_1)))$. Außerdem gilt $\text{gen}(\Gamma, R(\tau_1|C_1)) = \text{gen}(\Gamma, \tau_1|C_1)$ und da $C, \Gamma + [x : \text{gen}(\Gamma, \tau_1|C_1)]) \vdash M_2 : \tau$ eine gültige Typisierung ist, können wir die Induktionshypothese anwenden und erhalten gültige Typisierungen $SC, S(\Gamma + [x : \text{gen}(\Gamma, \tau_1)]) \vdash M_2 : S\tau$ und $SC, S\Gamma \vdash \textbf{let}\ x = M_1\ \textbf{in}\ M_2 : S\tau$. □

4.3 Typinferenz für eingeschränkte Typen

Das generische Deduktionssystem ist syntaxorientiert und kann daher auf einfache Weise in einen Algorithmus ($\mathcal{C}$) transformiert werden, der, instanziert mit der Definition der Regeln R_{var}, R_{abs} und R_{app}, den allgemeinsten Typ eines gegebenen Ausdrucks berechnet. Algorithmus $\mathcal{C}$ ist fast als trivial zu bezeichnen: seine einzige Aktivität besteht in der korrekten Instanzierung der Regeln des Deduktionssystems und dem Aufsammeln der daraus resultierenden Bedingungen in einer globalen Menge von Einschränkungen. Der Algorithmus verwendet eine Funktion *inst*, welche die gebundenen Typvariablen eines Typschemas durch neue, sonst nicht verwendete Typvariablen ersetzt, d.h. $inst(\forall\overline{\alpha_n}.\tau|C) = R(\tau|C)$ für eine Variablenumbenennung $R = \{\alpha_i \mapsto \beta_i\}$.

Lemma 4.21 ($\mathcal{C}$ ist wohldefiniert). *Für $(C, \tau) = \mathcal{C}[\![M]\!]\Gamma$ ist $C, \Gamma \vdash M : \tau$ eine ableitbare Typaussage.*

Beweis. (Induktion über M)

$\boxed{M \equiv x}$ Aufgrund der Definition von *inst* gilt $\tau_0|C_0 \prec \Gamma(x)$ und $\tau_1|C_1 \prec R_{var}(\tau_0)$ und für $C = C_0 \cup C_1$ gilt offensichtlich $C \Vdash C_0 \cup C_1$ und somit ist $C, \Gamma \vdash x : \tau_1$ eine ableitbare Typaussage.

$\boxed{M \equiv \lambda x.N}$ Es existieren τ_0, C_0 sodaß $\tau_0|C_0 = \mathcal{C}[\![N]\!](\Gamma + [x : \alpha])$. Nach Induktionsannahme ist $C_0, (\Gamma + [x : \beta]) \vdash N : \tau_0$ eine ableitbare Typisierung. Aufgrund der Definition von *inst* gilt $\tau_1|C_1 \prec R_{abs}(\beta, \tau_0)$ und mit $= C_0 \cup C_1$ gilt $C \Vdash C_0 \cup C_1$. Somit folgt mit Proposition 4.17 aber auch, daß $C, \Gamma + [x : \beta]) \vdash N : \tau$ eine ableitbare Typaussage ist und somit aus der [ABS]-Regel, daß auch $C, \Gamma \vdash \lambda x.N : \tau_1$ eine gültige Typaussage ist.

$\boxed{M \equiv M_1\, M_2}$ Nach Definition von $\mathcal{C}$ existieren eingeschränkte Typausdrücke $\tau_1|C_1 = \mathcal{C}[\![M_1]\!]\Gamma$, $\tau_1|C_1 = \mathcal{C}[\![M_2]\!]\Gamma$ und $\tau_3|C_3 = inst(R_{app}(\tau_1, \tau_2))$ Nach Induktionshypothese existieren gültige Typisierungen $C_1, \Gamma \vdash M_1 : \tau_1$ und $C_2, \Gamma \vdash M_2 : \tau_2$. Mit Proposition 4.17 sind damit aber auch für $C = C_0 \cup C_1 \cup C_2$ die Typaussagen$C, \Gamma \vdash M_1 : \tau_1$ und $C, \Gamma \vdash M_2 : \tau_2$ ableitbar. Mit $\tau_3|C_3 =\prec R_{app}(\tau_1, \tau_2)$ ist somit auch $C, \Gamma \vdash: M_1\, M_2$ ableitbar.

$\boxed{M \equiv \textbf{let}\ x = M_1\ \textbf{in}\ M_2}$ Nach Definition von $\mathcal{C}$ existieren $\tau_1|C_1 = \mathcal{C}[\![M_1]\!]\Gamma$ und

Algorithmus 4.1. ($\mathcal{C}$)

$\mathcal{C}[\![x]\!]\Gamma = \tau_1|C_0 \cup C_1$
wobei
$\tau_0|C_0 = \mathit{inst}(\Gamma(x))$
$\tau_1|C_1 = \mathit{inst}(R_{var}(\tau_0))$.

$\mathcal{C}[\![\lambda x.M]\!]\Gamma = \tau_1|C_0 \cup C_1$
wobei α „neu“ und
$\tau_0|C_0 = \mathcal{C}[\![M]\!](\Gamma + [x : \alpha])$
$\tau_1|C_1 = \mathit{inst}(R_{abs}(\alpha, \tau_0))$.

$\mathcal{C}[\![M_1\ M_2]\!]\Gamma = \tau_3|C_1 \cup C_2 \cup C_3$
wobei
$\tau_1|C_1 = \mathcal{C}[\![M_1]\!]\Gamma$
$\tau_2|C_2 = \mathcal{C}[\![M_2]\!]\Gamma$
$\tau_3|C_3 = \mathit{inst}(R_{app}(\tau_1, \tau_2))$.

$\mathcal{C}[\![\textbf{let } x = M_1 \textbf{ in } M_2]\!]\Gamma = \tau_2|C_1 \cup C_2$
wobei
$\tau_1|C_1 = \mathcal{C}[\![M_1]\!]\Gamma$ und
$\tau_2|C_2 = \mathcal{C}[\![M_2]\!](\Gamma + [x : \mathrm{gen}(\Gamma, \tau_1|C_1)])$.

$\tau_2|C_2 = \mathcal{C}[\![M_2]\!](\Gamma + [x : \mathrm{gen}(\Gamma, \tau|C_1)])$ und aufgrund der Induktionshypothese sind $C_1, \Gamma \vdash M_1 : \tau_1$ und $C_1, (\Gamma + [x : \mathrm{gen}(\Gamma, \tau|C_1)]) \vdash M_2 : \tau_2$ ableitbare Typaussagen. Für $C = C_1 \cup C_2$ folgt aus Proposition 4.17, daß auch $C, (\Gamma + [x : \mathrm{gen}(\Gamma, \tau|C_1)]) \vdash M_2 : \tau_2$ ableitbar ist, und da $C \Vdash C_2$ gilt, folgt insgesamt die Behauptung. □

Algorithmus $\mathcal{C}$ ist, bis auf die Benennung generischer Variablen, deterministisch. Es gilt

Proposition 4.22. *Sei $(\tau, C) = \mathcal{C}[\![M]\!]$ und S eine Substitution. Dann gibt es eine Variablenumbenennung R auf den von C durch inst-Aufrufe generierten Typvariablen, sodaß $S(R(\mathcal{C}[\![M]\!]\Gamma)) = \mathcal{C}[\![M]\!](S\Gamma)$.*

Beweis. Durch Induktion über M. □

Für den Beweis der Vollständigkeit von Algorithmus $\mathcal{C}$, benötigen wir einen Instanzbegriff für Typisierungen.

Definition 4.23. Eine Typisierung $C', \Gamma' \vdash M : \tau'$ heißt Instanz einer Typisierung $C, \Gamma \vdash M : \tau$, falls eine Substitution S existiert, sodaß

$$(1)\ C' \Vdash S(C)$$
$$(2)\ \Gamma' = S(\Gamma)$$
$$(3)\ \tau' = S(\tau)$$

Satz 4.24 ($\mathcal{C}$ ist vollständig). *Gegeben eine Instanz des generischen Typdeduktionssystems berechnet Algorithmus 4.1 für jeden Ausdruck M einen Repräsentanten aller möglichen Typisierungen von M unter einer gegebenen Typannahme Γ, d.h.: M ist typisierbar $\iff$ für $\tau | C = \mathcal{C}[\![M]\!]\Gamma$ gilt $\models C$.*

Beweis. Wir zeigen durch Induktion über M, daß für jede andere ableitbare Typisierung $C', \Gamma' \vdash M : \tau'$, für die $\Gamma'(x) \leq \Gamma(x)$ für alle $x \in dom(\Gamma)$ gilt, eine Substitution S existiert, sodaß $C' \Vdash S(C)$, $\Gamma' = S(\Gamma)$ und $\tau' = S(\tau)$ ist, d.h. $C', \Gamma' \vdash M : \tau'$ ist eine Instanz von $C, \Gamma \vdash M : \tau$.

$\boxed{M \equiv x}$ Nach Definition von $\mathcal{C}$ ist $\mathcal{C}[\![x]\!]\Gamma = \tau_1 | C_1 \cup C_2$ und es existiert ein τ_0, sodaß $\tau_0 | C_0 = inst(\Gamma(x))$, $\tau_1 | C_1 = inst(R_{var}(\tau_0))$. Nach Definition der ABS-Regel existieren $\tau'_o | C'_0 \prec \Gamma'(x)$, $\tau'_1 | C'_1 \prec R_{var}(\tau_0))$ und es gilt $C' \Vdash C'_0 \cup C'_1$. Da $\Gamma'(x) \leq \Gamma(x)$ gibt es eine Substitution S', sodaß $\forall \overline{\beta_m}.\tau'_x | C'_x = \Gamma'(x) = S'(\Gamma(x)) = S'(\forall \overline{\alpha_n}.\tau_x | C_x)$. Nach Definition von *inst* gibt es eine Variablenumbenennung $R_0 = \{\overline{\beta_m} \mapsto \overline{\beta'_m}$, sodaß $\tau_0 | C_0 = R_o(\tau_x | C_x)$. Nach Definition der generischen Instanz gibt es eine Substitution S'', sodaß $\tau'_0 = S''\tau'_x$ und $C'_o \Vdash S'C'_x$. Wegen $\tau'_x = S'\tau_x$ und $C'_x = S'C_x$ folgt $\tau'_0 = S''S'\tau_x$ und $C'_0 \Vdash S''S'C_x$. Nun gilt $R_{var}(\tau_0) = R_{var}(R_0\tau_x) = R_0(R_{var}(\tau_x)) = R_0(\forall \overline{\gamma_n}.\tau_v | C_v)$ und es existiert eine Variablenumbenennung $R_1 = \{\overline{\gamma_n} \mapsto \overline{\gamma'_n}\}$, sodaß $\tau_1 | C_1 = R_0 R_1(\tau_v | C_v)$. Da aber $R_{var}(\tau'_0) = R_{var}(S''S'\tau_x) = S''S'(R_{var}(\tau_x))$, folgt $\tau'_1 = S'''S''S'\tau_v$ und $C'_1 \Vdash S'''S''S'C_v$. Zu zeigen ist die Existenz einer Substitution Q, sodaß $\tau'_1 = Q\tau_1$ und $C' \Vdash Q(C_0 \cup C_1)$. Definiere

$$Q(\alpha) = \begin{cases} S'''S''S'(\beta) & \text{falls } R_0R_1(\beta) = \alpha,\ \beta \in dom(R_1 \circ R_0), \\ S'''S''S'(\alpha) & \text{sonst.} \end{cases}$$

Offensichtlich gilt $\tau_1 = R_0 R_1 \tau_x$ und damit folgt $\tau_1' = Q\tau_1$. Da $dom(R_1) \cap \mathcal{V}(C_x) = \emptyset$ gilt $R_1 C_x = C_x$ und daher $C_o = R_0 R_1 C_x$ sowie $C_1 = R_0 R_1 C_v$. Nach Konstruktion von Q gilt somit $QC_o = S'''S''S'C_x$ und $QC_1 = S'''S''S'C_v$ und da $dom(S''') \cap \mathcal{V}(S''S'C_x)) = \emptyset$ gilt $C_0' \Vdash QC_0$ und $C_1' \Vdash QC_1$. Wegen $C' \Vdash C_0' \cup C_1'$ folgt $C' \Vdash Q(C_o \cup C_1)$. Wegen $Q_{|\mathcal{FV}(\Gamma)} = S'$ folgt auch $\Gamma' = Q\Gamma$.

$\boxed{M \equiv \lambda x.N}$ Nach Definition von $\mathcal{C}$ existiert $\tau_0|C_0 = \mathcal{C}[\![N]\!](\Gamma + [x : \alpha]$ und nach Definition der ABS-Regel eine ableitbare Typaussage $C', \Gamma' + [x : \tau] \vdash N : \tau'$. Da nach Voraussetzung $\Gamma' \leq \Gamma$ existiert eine Substitution S' sodaß $\Gamma' = S'\Gamma$. Mit $S_\alpha = \{\alpha \mapsto \tau\}$ und $S'' = S' \circ S_\alpha$ gilt $\Gamma' + [x : \tau] \leq S''(\Gamma + [x : \alpha])$, da $\alpha \notin inv(S')$. Nun kann die Induktionshypothese angewendet werden, sodaß eine Substitution S existiert, mit $\tau' = S\tau_0$ und $C' \Vdash SC_0$. Nach Definition $\mathcal{C}$ existiert $\tau_1|C_1 = inst(R_{abs}(\alpha, \tau_0)) = R_0(\tau_a, C_a)$, wobei $R_0 = \{\overline{\beta_n} \mapsto \overline{\delta_n}\}$ eine Umbenennung der generischen Variablen von $R_{abs}(\alpha, \tau_0)$ ist. Nach Definition der ABS-Regel existiert $\tau''|C'' \prec R_{abs}(\tau, \tau')$. Betrachte $S_2 = S \circ S_\alpha$. Es gilt $S_2(\alpha) = \tau$, $S_2(\tau_0) = \tau'$, somit folgt $R_{abs}(\tau, \tau') = S_2(R_{abs}(\alpha, \tau_0))$, und $\tau'' = S_3 S_2 \tau_a$ sowie $C'' \Vdash S_3 S_2 C_a$. Wähle

$$Q(\alpha) = \begin{cases} S_3 S_2(\beta) & \text{falls } R_0(\beta) = \alpha,\ \beta \in dom(R_0), \\ S_3 S_2(\alpha) & \text{sonst.} \end{cases}$$

Offensichtlich gilt $Q\tau_1 = \tau''$, $C'' \Vdash QC_1$. Da $C' \Vdash C''$ folgt $C' \Vdash S_3 S_2 C_1$. Und da $(\{\alpha\} \cup dom(R_0)) \cap \mathcal{FV}(\Gamma) = \emptyset$ gilt auch $\Gamma' = Q\Gamma$.

$\boxed{M \equiv M_1\ M_2}$ Nach Definition von $\mathcal{C}$ existieren $\tau_1|C_1 = \mathcal{C}[\![M_1]\!]\Gamma$ sowie $\tau_2|C_2 = \mathcal{C}[\![M_2]\!]\Gamma$ und nach Definition der APP-Regel ableitbare Typaussagen $C', \Gamma' \vdash M_1 : \tau_1'$ und $C', \Gamma' \vdash M_2 : \tau_2'$. Nach Induktionshypothese existieren Substitutionen S_1, S_2 mit $\tau_1 = S_1\tau_1$, $\tau_2 = S_2\tau_2$, $C' \Vdash S_1 C_1$ und $C' \Vdash S_2 C_2$. Da $S_1\Gamma = S_2\Gamma = \Gamma'$ definieren wir eine Substitution

$$S_c(\alpha) = \begin{cases} S_1(\alpha) & \text{falls } \alpha \in dom(S_1) - \mathcal{FV}(\Gamma), \\ S_2(\alpha) & \text{falls } \alpha \in dom(S_2) - \mathcal{FV}(\Gamma), \\ S_2(\alpha) & \text{falls } \alpha \in \mathcal{FV}(\Gamma). \end{cases}$$

Offensichtlich gilt $\tau_1' = S_c\tau_1$, $\tau_2' = S_c\tau_2$, $C' \Vdash S_c C_1$ und $C' \Vdash S_c C_2$. Nach Definition der APP-Regel existieren $\tau''|C'' \prec R_{app}(\tau_1', \tau_2') = R_{app}(S_c\tau_1, S_c\tau_2) = S_c(R_{app}(\tau_1, \tau_2))$. Nach Definition von *inst* existiert daher eine Variablenumbenennung $R = \{\overline{\alpha_n} \mapsto$

$\overline{\beta_n}\}$ auf den generischen Variablen von $R_{app}(\tau_1, \tau_2) = \forall\overline{\alpha_n}.\tau_a|C_a$, sodaß $\tau_3|C_3 = R(\tau_a|C_a)$. Nach Definition von $\mathcal{C}$ existiert $\tau_3|C_3 = inst(R_{app}(\tau_1, \tau_2))$ und damit $\tau'' = S_i S_c \tau_a$ und $C'' \Vdash S_i S_c C_a$ für eine Substitution S_i. Wähle

$$Q(\alpha) = \begin{cases} S_i S_c(\beta) & \text{falls } R(\beta) = \alpha,\ \alpha \in dom(R), \\ S_i S_c(\alpha) & \text{sonst.} \end{cases}$$

Offensichtlich gilt $\tau'' = Q\tau_3$ und $C'' \Vdash QC_3$. Wegen $(dom(S_1) \cup dom(R)) \cap \mathcal{V}(C_1 \cup C_2) = \emptyset$ gilt $QC_1 = S_c C_1$ sowie $QC_2 = S_c C_2$ und mit $C' \Vdash C''$ folgt $C' \Vdash Q(C_1 \cup C_2 \cup C_3)$. Da $dom(S_i) \cap \mathcal{FV}(\Gamma) = \emptyset$ gilt $Q_{|\mathcal{FV}(\Gamma)} = S_{c|\mathcal{FV}(\Gamma)} = S_{2|\mathcal{FV}(\Gamma)}$ und durch Anwendung der Induktionsannahme $\Gamma' = S_2\Gamma$ und somit $\Gamma' = Q\Gamma$.

$\boxed{M \equiv \textbf{let } x = M_1 \textbf{ in } M_2}$ Nach Definition von $\mathcal{C}$ existieren $\tau_1|C_1 = \mathcal{C}[\![M_1]\!]\Gamma$, nach Definition der LET-Regel gibt es eine ableitbare Typisierung $C_1', \Gamma' \vdash M_1 : \tau_1'$ und aufgrund der Induktionsannahme existiert eine Substitution S_1 mit $\tau_1' = S_1\tau_1$, $C_1' \Vdash S_1 C_1$ und $\Gamma' = S_1\Gamma$. Nach Definition von $\mathcal{C}$ existiert $\tau_2|C_2 = \mathcal{C}[\![M_2]\!](\Gamma + [x : \sigma])$, wobei $\sigma = \text{gen}(\Gamma, \tau_1|C_1)$. Nach Definition der LET-Regel existiert eine ableitbare Typisierung $C', \Gamma' + [x : \sigma'] \vdash M_2 : \tau_2'$ mit $\sigma' = \text{gen}(\Gamma', \tau_1'|C_1')$ und $C' \Vdash C_1'$. Nun gilt aber $\sigma' = \text{gen}(S_1\Gamma, S_1\tau_1|S_1C_1)$ und da $C_1' \Vdash SC_1$ gilt $\sigma' \prec \sigma'' = \text{gen}(S_1\Gamma, S_1\tau_1|S_1C_1)$. Nach Proposition 4.15(ii) ist σ'' eine generische Instanz von $S_1(\text{gen}(\Gamma, \tau_1|C_1))$, d.h. $\sigma'' \prec S_1\sigma$, sodaß $\Gamma' + [x : S_1\sigma] \leq \Gamma + [x : \sigma]$. Nach Proposition 4.18 ist somit $C', \Gamma' + [x : S_1\sigma''] \vdash M_2 : \tau_2'$ ableitbar und erfüllt die Vorbedingungen der Induktionshypothese. Daher existiert eine Substitution S_2 mit $\tau_2' = S_2\tau_2$, $C' \Vdash S_2C_2$ und $S_2(\Gamma + [x : \sigma]) = \Gamma' + x : S_1\sigma''$. Zu zeigen ist die Existenz einer Substitution Q, mit $Q\Gamma = \Gamma'$, $Q\tau_2 = \tau_2'$ und $C' \Vdash Q(C_1 \cup C_2)$. Nun gilt $S_2\Gamma = S_1\Gamma$, sodaß wir Q wie folgt definieren:

$$Q(\alpha) = \begin{cases} S_2(\alpha) & \text{falls } \alpha \in \mathcal{FV}(\Gamma) \cup \mathcal{V}(\tau_2|C_2), \\ S_1(\alpha) & \text{falls } \alpha \in \mathcal{V}(\tau_1|C_1) - \mathcal{FV}(\Gamma). \end{cases}$$

Q ist wohldefiniert, offensichtlich gilt $\Gamma' = Q\Gamma$, $Q\tau_2 = \tau_2'$ sowie $Q(C_2) = S_2C_2$, $Q(C_1) = S_1(C_1)$ und daraus folgt sofort $C' \Vdash Q(C_1 \cup C_2)$. □

Aus der Korrektheit und Wohldefiniertheit von $\mathcal{C}$ folgt somit, daß eine strikte Trennung von Restriktionsauflösung und Typinferenz möglich ist. Dies war bis zur Ver-

öffentlichung dieses Resultats in [Kae92] nur für einfache Typinferenz ohne LET-Polymorphismus bekannt.[16] Darüber hinaus gilt:

Korrollar 4.25. *Typinferenz mit Prädikaten ist semi-entscheidbar, falls die Interpretationsfunktion für Prädikate berechenbar ist.*

Beweis. Aufgrund von Satz 4.24 ist M genau dann typisierbar unter Γ, wenn für die von Algorithmus 4.1 berechnete Prädikatmenge C eine erfüllende Substitution existiert. Sei $A = \{\alpha_1, \ldots, \alpha_n\}$ die Menge der in C vorkommenden Variablen. Wir nehmen o.B.d.A. an, daß F einen Typkonstruktor g mit Stelligkeit $\rho(g) = n$ enthält, ansonsten können wir einen neuen hinzufügen. Jedem Term $g(t_1, \ldots, t_n) \in T_F$ läßt sich eindeutig die Substitution $S_{t_1,\ldots,t_n} = \{\alpha_i \mapsto t_i\}$ zuordnen. Da die Menge aller Terme über F (d.h. $T_F(\emptyset)$) aufzählbar ist, gewinnt man auf diese Weise auch eine Aufzählung aller Substitutionen, bzw. aller Substitutionen die $S \models C$ erfüllen. Aus der Berechenbarkeit der Prädikate $(\hat{p})_{p \in P}$ folgt dann die Behauptung. □

4.4 Lösungen

Der Algorithmus zur Berechnung der allgemeinsten Typisierung ist natürlich in der Praxis nicht ausreichend, da er die Entscheidbarkeit der Typisierung nicht garantieren kann. Aber selbst wenn die Entscheidbarkeit für die Menge der Restriktionen gegeben wäre, etwa aufgrund der Eigenschaften der Prädikate, könnte man wohl kaum behaupten, daß der berechnete allgemeinste Typ eines Ausdrucks kompakt dargestellt würde. So hat beispielsweise der Ausdruck

$$\lambda f.\lambda x.f(fx)$$

im modifizierten Damas-Milner System den allgemeinsten Typ

$$\alpha \to \beta \to \gamma \mid \{\alpha = \beta \to \epsilon, \alpha = \epsilon \to \gamma\},$$

obwohl

$$(\beta \to \beta) \to \beta \to \beta \mid \emptyset$$

[16] Siehe z.B. [Wan87] oder [Bar92].

eine wesentliche kompaktere Darstellung des gleichen Typs ist. Der zweite Typ läßt sich aus dem ersten durch Lösen der Gleichungen mit Hilfe des Unifikationsalgorithmus, Anwenden des allgemeinsten Unifikators

$$\{\epsilon \mapsto \beta, \gamma \mapsto \beta, \alpha \mapsto \beta \to \beta\}$$

und Eliminieren der redundanten Gleichungen

$$\{\beta \to \beta = \beta \to \beta, \beta \to \beta = \beta \to \beta\}$$

gewinnen.

Definition 4.26. Eine Menge L von Lösungen für C ist *vollständig*, falls jede Lösung aus $R \in \mathcal{L}_P(C)$ Instanz einer Lösung aus L ist, d.h., falls $L \subseteq \mathcal{L}_P(C)$ und $\forall R \in \mathcal{L}_P(C) : \exists S \in L : S \leq^{\mathcal{V}(C)} R$. L ist eine vollständige Menge *minimaler* Lösungen, falls jedes Element von L minimal bzgl. $\leq^{\mathcal{V}(C)}$ ist, d.h., falls $\forall S_1, S_2 \in L : S_1 \not\leq^{\mathcal{V}(C)} S_2$.

Vollständige Mengen minimaler Lösungen müssen nicht existieren, falls sie existieren sind sie jedoch eindeutig bestimmt. Restriktionsauflösung kann in Übereinstimmung mit der Unifikationtheorie für Algebren mit nicht leerer Gleichungstheorie[17] folgendermaßen klassifiziert werden: Sei $\mu\mathcal{L}_P(C)$ eine vollständige Menge minimaler Lösungen. Restriktionsauflösung nennen wir

unitär	falls $\lvert\mu\mathcal{L}_P(C)\rvert \leq 1$ für alle endlichen C
finit	falls $\lvert\mu\mathcal{L}_P(C)\rvert < \infty$ für alle endlichen C
infinit	falls $\lvert\mu\mathcal{L}_P(C)\rvert = \infty$ für ein endliches C
nullstellig	falls $\mu\mathcal{L}_P(C)$ für ein endliches C nicht existiert
entscheidbar	falls $\mathcal{L}_P(C) = \emptyset$ entscheidbar für endliche C

Unglücklicherweise ist die Restriktionsauflösung sogar im einfachen Fall der parametrischen Überladungen sowie für die strukturelle Konversionstypisierung infinit. Beispielsweise gilt für $F_0 = \{i, r\}$, $F_1 = \{l\}$ und $P_1 = \{p\}$ mit der Definition $\hat{p}(t) \stackrel{\text{def}}{=} \exists k \in \mathbb{N} : t = l^k(i)$

$$\mu\mathcal{L}_P(\{p(\alpha)\}) = \{\{\alpha \mapsto l^k(i)\} \mid k \in \mathbb{N}\}\,.$$

[17] Siehe z.B. [Sie89].

Desweiteren besitzt die Prädikatmenge $\{q(\alpha, \beta)\}$ für $P_2 = \{q\}$ mit der Definition

$$\hat{q}(t_1, t_2) \stackrel{\text{def}}{=} t_1 = t_2 \vee \exists k \in \mathbb{N} : t_1 = l^k(i) \wedge t_2 = l^k(r)$$

die vollständige Menge minimaler Lösungen

$$\mu\mathcal{L}_P(\{q(\alpha, \beta)\}) = \{\{\alpha \mapsto \beta\}\} \cup \{\{\alpha \mapsto l^k(i), \beta \mapsto l^k(r)\} \mid k \in \mathbb{N}\}\,.$$

Daher scheint es nicht angebracht zu sein, vollständige Mengen minimaler Lösungen als Repräsentation des allgemeinsten Typs zu wählen.[18]

4.5 Vereinfachungen

Für unitäre Unifikationsprobleme ist der allgemeinste Unifikator immer eine vereinfachende Substitution einer gegebene Gleichungsmenge: Variablenbelegungen werden soweit als möglich bestimmt, ohne die Menge der möglichen Lösungen einzuschränken.

Definition 4.27. Eine Substitution S heißt *Vereinfachung* von C (S *ve* C), falls für jede Substitution L mit $L \models C$ eine Substitution L' existiert, sodaß $L = L' \circ S$.

Vereinfachende Substitutionen müssen eine gegebene Restriktionsmenge nicht wirklich vereinfachen. So sind z.B. die identische Substitution sowie Variablenumbenennungen vereinfachende Substitutionen.

Korrollar 4.28. *Für jede Vereinfachung S von C läßt sich jede Lösung Q von C in der Form $S' \circ S$ darstellen.*

Beweis. Da Q eine Lösung von C ist, gibt es eine Restsubstitution L mit der Eigenschaft $L \circ Q \models C$. Falls nun kein S' existiert, das $Q = S' \circ S$ erfüllt, dann gibt es ein α mit $S(\alpha) \not\leq Q(\alpha)$ und somit gilt auch $S(\alpha) \not\leq L(Q(\alpha))$. Dies steht jedoch im Widerspruch dazu, daß sich jede erfüllende Substitution in der Form $L' \circ S$ darstellen läßt. □

Lemma 4.29. S *ve* C, S' *ve* $S(C) \iff S' \circ S$ *ve* C

[18] Außer für den Fall, daß eine unitäre Unifikation gegeben ist.

Beweis. "⇒": Da S eine Vereinfachung von C ist, kann jede erfüllende Substitution $L \models C$ in der Form $L' \circ S$ notiert werden. Da $L' \models S(C)$ und S' *ve* $S(C)$ gibt es eine Substitution L'', sodaß $L' = L'' \circ S'$ und somit $L = L'' \circ S' \circ S$.

"⇐": Wegen $L \models C \Rightarrow L = L'' \circ S' \circ S$ folgt mit $L' = L'' \circ S'$ sofort S *ve* C. Andererseits gilt mit $L'' \circ S' \circ S \models C$ auch $L'' \models (S' \circ S)C$ und somit S' *ve* $S(C)$. □

Lemma 4.30 (Diamantlemma für Vereinfachungen). *Für lösbare Restriktionsmengen gilt:*[19]

$$S_1 \text{ ve } C,\ S_2 \text{ ve } C \Rightarrow \exists R_1 \text{ ve } S_1(C),\ R_2 \text{ ve } S_2(C) : R_1 \circ S_1 = R_2 \circ S_2$$

Beweis. Sei L eine Lösung für C. Da S_1 und S_2 Vereinfachungen von C sind, existieren Substitutionen L_1, L_2 mit $L = L_1 \circ S_1$ und $L = L_2 \circ S_2$. Betrachte für $D = \{\alpha_1, \ldots, \alpha_n\} = dom(S_1) \cap dom(S_2)$ und $t = (\alpha_1, \ldots, \alpha_n)$ die Terme $t_1 = S_1(t)$ bzw. $t_2 = S_2(t)$. Da $L_1(t_1) = L_2(t_2)$ existiert der allgemeinste Unifikator u von t_1 und t_2 und Substitutionen L_1' bzw. L_2', sodaß $L_{1|D} = L_1 \circ u$ und $L_{2|D} = L_2 \circ u$. Man überzeugt sich leicht, daß für

$$R_1(\alpha) = \begin{cases} u(S_1(\alpha)), & \text{falls } \alpha \in D \\ S_2(\alpha), & \text{falls } \alpha \in dom(S_2) - D \\ \alpha, & \text{sonst} \end{cases}$$

$$R_2(\alpha) = \begin{cases} u(S_1(\alpha)), & \text{falls } \alpha \in D \\ S_1(\alpha), & \text{falls } \alpha \in dom(S_1) - D \\ \alpha, & \text{sonst} \end{cases}$$

die Behauptung erfüllt ist. □

Im allgemeinen gibt es viele vereinfachende Substitutionen für eine gegebene Prädikatmenge C. So ist beispielsweise die identische Substitution eine Vereinfachung für jede Menge C, und das gleiche gilt auch für beliebige Variablenumbenennungen.

Definition 4.31. Eine Substitution S heißt *genaueste Vereinfachung* eines Rstriktionsproblems C (S *gve* C), falls für jede andere Vereinfachung S' eine Substitution S'' existiert, sodaß $S = S'' \circ S'$.

[19] Siehe Abbildung 4.9

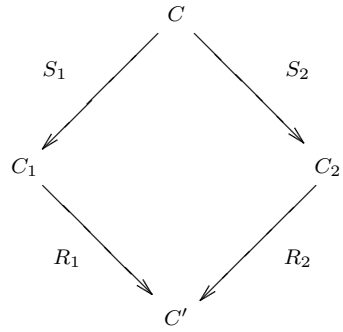


Abbildung 4.9: Diamantlemma für Vereinfachungen

Korrollar 4.32. *S ve C, S' gve $S(C) \Rightarrow S' \circ S$ gve C*

Beweis. Zu zeigen ist, daß für jede Vereinfachung R von C eine Substitution R' existiert, die $S' \circ S = R' \circ R$ erfüllt. Aus Lemma 4.30 folgt die Existenz zweier Vereinfachungen Q *ve* $S(C)$, Q' *ve* $R(C)$ mit $Q \circ S = Q' \circ R$. Da S' eine genaueste Vereinfachung von $S(C)$ ist, gibt es eine Substitution Q'' mit $S' = Q'' \circ Q$. Dann ist aber auch $S' \circ S = Q'' \circ Q' \circ R$ □

Die Menge der Vereinfachungen einer Prädikatmenge C ist bezüglich der Instanzrelation partiell geordnet, eine genaueste Vereinfachung ist, falls sie existiert, ein minimales Element dieser Ordnung. Über die Existenz bzw. Berechenbarkeit einer genauesten Vereinfachung läßt sich natürlich ohne eine Kenntnis der Interpretation für P keine Aussage treffen.

Lemma 4.33. *Die identische Substitution ist genau dann eine genaueste Vereinfachung von C, falls die beiden folgenden Bedingungen erfüllt sind:*

$$(a)\ \forall \alpha, \beta \in \mathcal{V}(C) : \exists L \models C : L(\alpha) \neq L(\beta)$$
$$(b)\ \forall \alpha \in \mathcal{V}(C) : \exists L_1, L_2 \models C : root(L_1(\alpha)) \neq root(L_2(\alpha))$$

Beweis. Laut Definition von *gve* ist die identische Substitution Id genau dann eine genaueste Vereinfachung von C, wenn $\forall S \; ve \; C : \exists S' : Id = S' \circ S$ gilt.

"$\Rightarrow$": Nehmen wir an es gäbe $\alpha, \beta \in \mathcal{V}(C)$, sodaß $\forall L \models C : L(\alpha) = L(\beta)$. Dann ist $\{\alpha \mapsto L(\alpha)\}$ eine Vereinfachung von C. Da $L(\alpha) = t \in T_F$, kann es kein S geben, das $S(t) = \alpha$ erfüllt. Falls ein $\alpha \in \mathcal{V}(C)$ existiert, sodaß $\forall L_1, L_2 \models C : L_1(\alpha) = L_2(\alpha) = f(t_1, \ldots, t_n)$, dann ist für $\{\alpha_1, \ldots, \alpha_n\} \subseteq \mathcal{V} - \mathcal{V}(C)$ die Substitution $R = \{\alpha \mapsto f(\alpha_1, \ldots, \alpha_n)\}$ eine Vereinfachung von C. Offensichtlich gibt es kein S mit $Id = S \circ R$.

"$\Leftarrow$": Sei S eine Vereinfachung von C. Zunächst einmal gilt $cod(S) \subseteq \mathcal{V}$, denn sonst gäbe es ein $\alpha \in \mathcal{V}$ und ein $f \in F_n$, $n \geq 0$ derart, daß $S(\alpha) = f(\tau_1, \ldots, \tau_n)$. Wegen $S \; ve \; C$ gilt dann $\forall L \models C : L(\alpha) = S'(S(\alpha)) = S'(f(\tau_1, \ldots, \tau_n)) = f(S'(\tau_1), \ldots, S'(\tau_n))$. Dies impliziert $\forall L \models C : root(S(\alpha)) = f$, im Widerspruch zu (b). Bedingung (a) impliziert, daß für je zwei Variablen $\alpha, \beta \in \mathcal{V}(C)$ eine Lösung $L = S' \circ S$ existiert, die $S'(S(\alpha)) \neq S'(S(\beta))$ erfüllt. Dies kann nur für $S(\alpha) \neq S(\beta)$ geschehen, also gilt $\forall \alpha, \beta \in \mathcal{V}(C) : S(\alpha) \neq S(\beta)$. Wegen $S \in \mathcal{V} \to \mathcal{V}$ folgt daraus $Id = S' \circ S$. □

Lemma 4.34. *Genaueste Vereinfachungen einer lösbaren Prädikatmenge sind eindeutig bestimmt (modulo $\cong$).*

Beweis. Seien S_1, S_2 genaueste Vereinfachungen von C. Dann gilt $S_1 = Q \circ S_2$ und $S_2 = R \circ S_1$, d.h. $S_1 \leq S_2$ und $S_2 \leq S_1$ und somit $S_1 \cong S_2$. □

Lemma 4.35. *Falls $\mu\mathcal{L}_P(C) \neq \emptyset$ dann ist $\bigwedge \mu\mathcal{L}_P(C)$ eine genaueste Vereinfachung von C.*

Beweis. Folgt aus der Verbandsstruktur der Vereinfachungen. □

Lemma 4.36. *Sei C eine Prädikattermmenge und $S_1 \ldots S_n$ Substitutionen, sodaß*

$$(1) \models C \iff \exists i \in 1..n : \models S_i(C)$$
$$(2) \; \forall i \in 1..n : \models S_i(C)\,.$$

Seien $A_1, \ldots, A_n$ genaueste Vereinfachungen von $S_1(C), \ldots, S_n(C)$. Dann ist $\bigwedge_{i=1..n} S_{i|\mathcal{V}(C)}$ eine Vereinfachung und $\bigwedge_{i=1..n}(A_i \circ S_i)_{|\mathcal{V}(C)}$ eine genaueste Vereinfachung von C.

Beweis. $\bigwedge_{i=1..n} S_i$ ist eine Vereinfachung von C, da $(\bigwedge_{i=1..n} S_i) \leq S_i$ für alle $i \in 1..n$, was aufgrund von Bedingung (1) auch $(\bigwedge_{i=1..n} S_i) \leq L$ für alle $L \models C$ impliziert. Wegen $(Q \wedge R) \circ S = Q \circ S \wedge R \circ S$ und $A_i = \bigwedge_{L \in \mathcal{L}(C_i)} L$ gilt

$$\begin{aligned}
\textstyle\bigwedge_{i=1}^{n}(A_i \circ S_i) &= \textstyle\bigwedge_{i=1}^{n}((\bigwedge_{L \in \mathcal{L}(C_i)} L) \circ S_i) \\
&= \textstyle\bigwedge_{i=1}^{n}(\bigwedge_{L \in \mathcal{L}(C_i)} L \circ S_i) \\
&= \textstyle\bigwedge_{\substack{i=1 \\ L \in \mathcal{L}(C)}}^{n} (L \circ S_i) \\
&= \textstyle\bigwedge_{\substack{i=1 \\ L \in \mathcal{L}(C)}}^{n} L
\end{aligned}$$

somit ist $\bigwedge_{i=1..n}(A_i \circ S_i)$ eine genaueste Vereinfachung von C. □

Lemma 4.37. *Sei $C, \Gamma \vdash M : \tau$ eine gültige Typisierung und S eine vereinfachende Substitution, dann ist auch $SC, S\Gamma \vdash M : S\tau$ eine gültige Typisierung.*

Beweis. Daß $SC, S\Gamma \vdash M : S\tau$ eine ableitbare Typisierung ist, folgt aus der Abgeschlossenheit von ableitbaren Typaussagen unter Substitutionen (Lemma 4.20). Da S eine Vereinfachung von C ist und C lösbar, ist natürlich auch $S(C)$ lösbar. □

Offensichtlich gilt:

Proposition 4.38. *Sei $C = C_1 \cup C_2$ und es gelte $\mathcal{V}(C_1) \cap \mathcal{V}(C_2) = \emptyset$. Dann gilt für jede Lösung von $S \in \mathcal{L}_P(C)$: $S_{|\mathcal{V}(C)} = (S_1 \circ S_2)_{|\mathcal{V}(C)}$, für $S_1 \in \mathcal{L}_P(C_1)$ und $S_2 \in \mathcal{L}_P(C_2)$.*

Vereinfachende Substitutionen können benutzt werden, um den von Algorithmus $\mathcal{C}$ berechneten allgemeinsten Typ zu vereinfachen, sie können jedoch auch zur Verbesserung von Algorithmus $\mathcal{C}$ eingesetzt werden.

Lemma 4.39. *Sei $C, \Gamma \vdash M : \tau$ eine ableitbare Typaussage $C', \Gamma' \vdash M : \tau'$ eine Instanz von $C, \Gamma \vdash M : \tau$ und S eine Vereinfachung von C. Dann gibt es für jede Lösung $L \models C'$ eine Substitution S' mit der Eigenschaft $L = S' \circ S$.*

Beweis. Folgt direkt aus Korrollar 4.28. □

Allerdings gilt nicht notwendigerweise, daß $C', \Gamma' \vdash M : \tau'$ eine Instanz von $SC, S\Gamma \vdash M : S\tau$ ist, wie das folgende Gegenbeispiel zeigt: Wähle $\tau = \tau'$, $C = C' = \{\alpha = \beta\}$ und $S = \{\alpha \mapsto \beta\}$. Offensichtlich ist $C' = \{\alpha = \beta\}$ keine Instanz von $S(C) = \{\beta = \beta\}$.

4.6 Algorithmus $\mathcal{D}$

Wir stellen nun eine Verbesserung von Algorithmus $\mathcal{C}$ vor, der sich auf die Anwendung vereinfachender Substitutionen stützt. Grundidee dieses Algorithmus ist die Verschränkung der Vereinfachung von Prädikatmengen mit dem Aufsammeln der durch die Deduktionsregeln implizierten Einschränkungen. Diese Strategie führt i.a. zu einer Reduktion der generischen Variablen und der Anzahl Prädikatterme, die let-gebundenen Bezeichnern im Kontext zugeordnet werden. Dies wiederum hat zur Folge, daß identische Einschränkungen bei der Instanzierung des Typs polymorpher Bezeichner an ihren Verwendungsstellen nicht wiederholt instanziert werden und infolgedessen natürlich nicht mehrfach gleichartig vereinfacht werden müssen. In der Regel wird daher Algorithmus $\mathcal{D}$ den gesuchten Typ schneller berechnen, als Algorithmus $\mathcal{C}$ gefolgt von einer Vereinfachung der Restriktionsmengen. Dazu nehmen wir an, daß für die betrachteten Restriktionen eine Funktion *simplify* existiert, die angewendet auf einen gegebenen eingeschränkten Typ $\tau|C$ eine vereinfachende, idempotente Substitution S und einen vereinfachten eingeschränkten Typ $\tau'|C'$ liefert, sodaß $\tau' = S(\tau)$, $S(C') = C'$, $C' \equiv S(C)$ und $\mathcal{FV}(C') \supseteq \mathcal{FV}(C)$ gilt. Falls *simplify* fehlschlägt, gibt es keine Lösung für C.[20]

Man sieht leicht, daß man für den Fall des rein parametrischen Polymorphismus aus Algorithmus $\mathcal{D}$ den Algorithmus $\mathcal{W}$ erhält, wenn man die Funktion *simplify* wie folgt definiert:

$$\mathit{simplify}(\tau, C) = (\mu, \mu(\tau|C)), \qquad \text{falls } \mu = \mathrm{mgu}(C)$$

Existiert kein Unifikator für C, so schlägt *simplify* fehl.[21]

Satz 4.40. *Sei* $(S, \tau|C) = \mathcal{D}[\![M]\!]\Gamma$ *und* $\tau'|C' = \mathcal{C}[\![M]\!]\Gamma'$ *und es gelte* $\Gamma \equiv_{\prec} \Gamma'$ *mit* $\mathcal{FV}(\Gamma') \supseteq \mathcal{FV}(\Gamma)$*, dann gibt es eine Variablenumbenennung* R *mit* $dom(R) \cap$

[20] simplify kann durchaus nichtdeterministisch sein, solange die obigen Bedingungen erfüllt sind.

[21] Die gleiche Argumentation kann natürlich auf beliebige, unitär unifizierende Algebren angewendet werden.

Algorithmus 4.2. ($\mathcal{D}$)

$\mathcal{D}[\![x]\!]\Gamma = (S, \tau|C)$
falls eine Substitution S existiert, sodaß
$\tau_0|C_0 = \mathit{inst}(\Gamma(x))$
$\tau_1|C_1 = \mathit{inst}(R_{var}(\tau_0))$
$(S, \tau|C) = \mathit{simplify}(\tau_1|C_0 \cup C_1)$.

$\mathcal{D}[\![\lambda x.M]\!]\Gamma = (S_2 \circ S_1, \tau|C)$
falls Substitutionen S_1, S_2 und α „neu“ existieren, sodaß
$(S_1, \tau_1|C_1) = \mathcal{D}[\![M]\!](\Gamma + [x : \alpha])$
$\tau_2|C_2 = \mathit{inst}(R_{abs}(S_1\alpha, \tau_1))$
$(S_2, \tau|C) = \mathit{simplify}(\tau_2|C_1 \cup C_2)$.

$\mathcal{D}[\![M_1\, M_2]\!]\Gamma = (S_3 \circ S_2 \circ S_1, \tau|C)$
falls Substitutionen S_1, S_2, S_3 existieren, sodaß
$(S_1, \tau_1|C_1) = \mathcal{D}[\![M_1]\!]\Gamma$,
$(S_2, \tau_2|C_2) = \mathcal{D}[\![M_2]\!]S_1\Gamma$ und
$\tau_3|C_3 = \mathit{inst}(R_{app}(S_2\tau_1, \tau_2))$
$(S_3, \tau|C) = \mathit{simplify}(\tau_3|S_2C_1 \cup C_2 \cup C_3)$.

$\mathcal{D}[\![\textbf{let } x = M_1 \textbf{ in } M_2]\!]\Gamma = (S_3 \circ S_2 \circ S_1, \tau|C)$
falls Substitutionen S_1,S_2,S_3 existieren, sodaß
$(S_1, \tau_1|C_1) = \mathcal{D}[\![M_1]\!]\Gamma$ und
$(S_2, \tau_2|C_2) = \mathcal{D}[\![M_2]\!](S_1\Gamma + [x : \mathrm{gen}(S_1\Gamma, \tau_1|C_1)])$
$(S_3, \tau|C) = \mathit{simplify}(\tau_2|S_2C_1 \cup C_2)$.

In allen anderen Fällen schlägt der Algorithmus fehl.

$\mathcal{FV}(\Gamma) = \emptyset$ und $\mathit{dom}(R) \cap \mathit{inv}(S) = \emptyset$, sodaß $S(\tau|C) = \tau|C$, $S(R(\tau')) = \tau$ und $C \equiv S(R(C'))$. Falls $\mathcal{D}$ fehlschlägt, dann ist M nicht typisierbar, sonst ist $C, S(\Gamma) \vdash M : \tau$ eine ableitbare Typisierung und SR eine Vereinfachung von C'.

Beweis. Wir zeigen zunächst, daß die Ableitbarkeit von $C, S(\Gamma) \vdash M : \tau$ aus den restlichen Aussagen folgt, die anschließend durch Induktion über M gezeigt werden.

Sei also $\tau'|C' = \mathcal{C}[\![M]\!]\Gamma'$, dann ist $C', \Gamma' \vdash M : \tau'$ ableitbar. Sei $(S, \tau|C) = \mathcal{D}[\![M]\!]\Gamma$, dann gilt $S(R(\tau')) = \tau$, $S(C) = C$ und $C \equiv S(R(C'))$. Da ableitbare Typaussagen unter Substitution abgeschlossen sind, folgt die Ableitbarkeit von $S(R(C')), S(R(\Gamma')) \vdash M : S(R(\tau'))$. Da nach Voraussetzung $dom(R) \cap \mathcal{FV}(\Gamma)) = \emptyset$ folgt die Ableitbarkeit von $S(R(C')), S(\Gamma') \vdash M : \tau$. Nun gilt aber auch $C \equiv S(R(C'))$ und somit folgt aus Korrollar 4.19 die Ableitbarkeit von $C, S(\Gamma) \vdash M : \tau$.

$\boxed{M \equiv x}$ Sei $\tau'|C' = \mathcal{C}[\![x]\!]\Gamma'$ und $(S, \tau|C) = \mathcal{D}[\![x]\!]\Gamma$. Dann gibt es eine Variablenumbenennung R, sodaß $R(\tau_1'|C_1' \cup C_2') = \tau_1|C_1 \cup C_2$, da die ersten beiden Zeilen der Algorithmen $\mathcal{C}$ und $\mathcal{D}$ identisch sind und das Ergebnis sich daher höchstens in den generierten Instanzvariablen unterscheidet. Offensichtlich gilt $dom(R) \cap \mathcal{FV}(\Gamma) = \emptyset$ und $dom(R) \cap dom(S) = \emptyset$. Sei $(S, \tau|C) = simplify(\tau_1|C_0 \cup C_1)$. Nach Definition von *simplify* ist S eine Vereinfachung von $C_0 \cup C_1$ und es gilt $S(\tau) = \tau = S(\tau_1) = S(R(\tau_1'))$, $S(C) = C$ und $C \equiv S(C_1 \cup C_2)$ und damit auch $C \equiv S(R(C_1' \cup C_2'))$.

$\boxed{M \equiv \lambda x.N}$ Nach Definition von $\mathcal{D}$ existiert $(S_1, \tau_1|C_1) = \mathcal{D}[\![N]\!](\Gamma' + [x : \alpha])$. Schlägt $\mathcal{D}$ fehl, so ist N nach Induktionsannahme nicht typisierbar. Nach Definition von $\mathcal{C}$ existiert $\tau_1'|C_1' = \mathcal{C}[\![N]\!](\Gamma + [x : \alpha'])$. Wähle nun $Q = \{\alpha' \mapsto \alpha\}$, dann existiert aufgrund von Proposition 4.22 eine Variablenumbenennung R_0, sodaß

$$QR_0(\mathcal{C}[\![N]\!](\Gamma' + [x : \alpha'])) = \mathcal{C}[\![N]\!](Q\Gamma' + [x : Q\alpha']) = (\tau_1''|C_1'')$$

d.h $QR_0(\tau_1'|C_1') = (\tau_1''|C_1'')$. Es gilt

$$Q\Gamma' + [x : Q\alpha'] = \Gamma' + [x : \alpha] \equiv_{\prec} \Gamma + [x : \alpha]$$

und somit folgt aus der Induktionshypothese die Existenz einer Variablenumbenennung R_1', sodaß

$$\tau_1|C_1 = S_1R_1'QR_0(\tau_1'|C_1').$$

Mit $R_1 = R_1'Q$ folgt $\tau_1|C_1 = S_1R_1(\tau_1'|C_1')$, wobei R_1 offensichtlich eine Variablenumbenennung ist. Nach Definition von $\mathcal{C}$ existiert $\tau_2'|C_2' = inst(R_{abs}(\alpha', \tau_1'))$. Nach Definition von $\mathcal{D}$ existiert $\tau_2|C_2 = inst(R_{abs}(S_1\alpha, \tau_1))$. Wegen $R_1(\alpha') = \alpha$ und $\tau_1 = S_1R_1\tau_1'$ gilt $\tau_2|C_2 = inst(R_{abs}(S_1R_1\alpha', S_1R_1\tau_1')) = S_1R_1(inst(R_{abs}(\alpha', \tau_1')))$, sodaß eine Variablenumbenennung U existiert, mit $\tau_2|C_2 = S_1R_1U(\tau_2'|C_2')$. Nach Definition $\mathcal{D}$ gilt $(S_2, \tau|C) = simplify(\tau_2|C_1 \cup C_2)$, mit $\tau = S_2(\tau)$ und $C \equiv S_2(C_1 \cup C_2)$. Zu zeigen ist $\tau = S_2S_1R\tau_2'$ und $C \equiv S_2S_1R(C_1' \cup C_2'))$ für $R = R_1 \circ U$. Da

$\tau_2 = S_2R_1U\tau_2'$ folgt sofort $\tau = S_2S_1R\tau_2'$. Da $C_2 = S_1R_1C_1'$ und $C_1 = S_1R_1C_1'$ folgt $C \equiv S_2S_1R_1(C_1' \cup C_2'))$. Nun involviert aber U keine Variablen in $C_1' \cup C_2'$ sodaß auch $C \equiv S_2S_1R(C_1' \cup C_2')$ gilt.

$\boxed{M \equiv M_1\, M_2}$ Nach Definition von $\mathcal{C}$ existiert $\tau_1'|C_1' = \mathcal{C}[\![M_1]\!]\Gamma'$ und nach Definition von $\mathcal{D}$ ex. $(S_1, \tau_1|C_1) = \mathcal{C}[\![M_1]\!]\Gamma$. Aus der Induktionshypothese folgt $\tau_1 = S_1R_1\tau_1'$, $C_1 \equiv S_1R_1C_1'$ und $S_1(\tau_1|C_1) = (\tau_1|C_1)$. Nach Definition $\mathcal{D}$ ex. $(S_2, \tau_2|C_2) = \mathcal{D}[\![M_2]\!]S_1\Gamma$. Betrachte $(\tau_2', C_2') = \mathcal{C}[\![M_2]\!]\Gamma'$. Aufgrund von Proposition 4.22 existiert eine Variablenumbenennung R_0 mit $S_1R_o(\mathcal{C}[\![M_2]\!]\Gamma') = \mathcal{C}[\![M_2]\!]S_1\Gamma' = (\tau_2'', C_2'')$. Damit kann man die Induktionshypothese anwenden und erhält $\tau_2 = S_2R_2\tau_2''$, $C_2 \equiv S_2R_2C_2''$ und also $\tau_2 = S_2R_2S_1R_o\tau_2'$, und $C_2 \equiv S_2R_2S_1R_oC_2'$. Es gilt $dom(R_2) \cap inv(S_1) = \emptyset$ und somit $\tau_2 = S_2S_1R_2R_o\tau_2'$ und $C_2 \equiv S_2S_1R_2R_oC_2'$. Nach Definition von $\mathcal{D}$ existieren $\tau_3|C_3$ mit

$$\begin{aligned}\tau_3|C_3 &= inst(R_{abs}(S_2\tau_1, \tau_2)) \\ &= inst(R_{abs}(S_2S_1R_1\tau_1', S_2S_1R_2R_0\tau_2')) \\ &= S_2S_1(inst(R_{abs}(R_1\tau_1', R_2R_0\tau_2')))\end{aligned}$$

und da auch $dom(R_1) \cap \mathcal{V}(\tau_2') = dom(R_2) \cap \mathcal{V}(\tau_1') = \emptyset$

$$= S_2S_1(inst(R_{abs}(R_2R_0R_1\tau_1', R_2R_0R\tau_2')))$$

und es existiert nach Definition von *inst* eine Variablenumbenennung R_3, sodaß

$$\tau_3|C_3 = S_2S_1R_3R_2R_0R_1(\tau_3'|C_3').$$

Nach Definition von *simplify* existiert $(S_3, \tau|C)$ mit $S_3\tau = \tau = S_3\tau_3$, $S_3C = C$, und $C \equiv S_3(S_2C_1 \cup C_2 \cup C_3)$. Zu zeigen ist: es existiert ein R mit a) $\tau = S_3S_2S_1R\tau_3'$ und b) $C \equiv S_3S_2S_1R(C_1' \cup C_2' \cup C_3')$. Mit $R = R_3R_2R_0R_1$ folgt wg. $\tau_3 = S_2S_1R\tau_3'$ sofort a). Da $C_1 \equiv S_1R_1C_1'$ und $C_2 \equiv S_2S_1R_2R_0C_2'$ und $C_3 \equiv S_2S_1RC_3'$ folgt $C \equiv S_3(S_2S_1R_1C_1' \cup S_2S_1R_2R_0C_2' \cup S_2S_1R_1C_3')$, was identisch ist mit $S_3S_2S_1(R_1C_1' \cup R_2R_0C_2' \cup RC_3')$. Nun gilt aber $R_1C_1' = RC_1$ und $RC_2' = R_2R_0C_2'$ und damit folgt Teil b).

$\boxed{M \equiv \textbf{let}\ x = M_1\ \textbf{in}\ M_2}$ Nach Definition von $\mathcal{C}$ existiert $\tau_1'|C_1' = \mathcal{C}[\![M_1]\!]\Gamma'$, nach Definition von $\mathcal{D}$ existiert $(S_1, \tau_1) = \mathcal{D}[\![M]\!]\Gamma$. Da $\Gamma' \equiv_\prec \Gamma$ nach Voraussetzung, existiert nach Induktionshypothese eine Variablenumbenennung R_1 sodaß $\tau_1 = S_1R_1(\tau_1')$ und

$C_1 \equiv S_1R_1(C_1')$. Nach Definition von $\mathcal{D}$ existiert $(S_2, \tau_2|C_2) = \mathcal{D}[\![M_2]\!](\Gamma + [x : \text{gen}(S_1\Gamma, \tau_1|C_1)]$. Wir betrachten nun den Ausdruck $(\tau_2'|C_2') = \mathcal{C}[\![M_2]\!](\Gamma' + [x : \text{gen}(\Gamma', \tau_1'|C_1')])$. Aufgrund von Proposition 4.22 existiert eine Variablenumbenennung R', sodaß

$$S_1R_1R'(\tau_2'|C_2') = \mathcal{C}[\![M_2]\!](S_1R_1(\Gamma') + [x : S_1R_1(\text{gen}(\Gamma', \tau_1'|C_1'))])$$

aufgrund von Proposition. 4.15(iv)

$$= \mathcal{C}[\![M_2]\!](S_1R_1(\Gamma') + [x : S_1(\text{gen}(R_1\Gamma', R_1(\tau_1'|C_1')))])$$

und Proposition. 4.15(iii)

$$= \mathcal{C}[\![M_2]\!](S_1R_1(\Gamma') + [x : \text{gen}(S_1R_1(\Gamma'), S_1R_1(\tau_1'|C_1'))])$$

und da $dom(R_1) \cap \mathcal{FV}(\Gamma') = \emptyset$

$$= \mathcal{C}[\![M_2]\!](S_1\Gamma' + [x : \text{gen}(S_1\Gamma', S_1R_1(\tau_1'|C_1'))])$$

und wegen $\tau_1 = S_1R_1(\tau_1')$

$$= \mathcal{C}[\![M_2]\!](S_1\Gamma' + [x : \text{gen}(S_1\Gamma', \tau_1|S_1R_1(C_1'))])$$

Nun gilt $C_1 \equiv S_1R_1(C_1')$ und damit

$$S_1\Gamma + [x : \text{gen}(S_1\Gamma, \tau_1|C_1)] \equiv_{\prec} S_1\Gamma' + [x : \text{gen}(S_1\Gamma', \tau_1|S_1R_1C_1')]$$

sodaß wir aufgrund der Induktionshypothese die folgenden Aussagen erhalten: $\tau_2 = S_2R_2S_1R'(\tau_2')$ sowie $C_2 \equiv S_2R_2S_1R_1R'(C_2')$. Nach Definition von *simplify* gilt $\tau = S_3(\tau_2)$ und $C \equiv S_3(S_2C_1 \cup C_2)$. Somit gilt $\tau = S_3S_2S_1R_1R'(\tau_2')$ und da $dom(R_2) \cap inv(S_3 \circ S_2 \circ S_1) = \emptyset$ auch $\tau = S_3S_2S_1R_2R_1R'(\tau_2')$. Da auch $dom(R_2) \cap (\mathcal{V}(C_1) \cup rng(R_1)) = \emptyset$ gilt $R_1(C_1') = R_2R_1(C_1')$. Mit $R = R_2 \circ R_1 \circ R'$ und $S = S_3 \circ S_2 \circ S_1$ folgt $\tau = SR(\tau_2')$ und $C \equiv SR(C_1' \cup C_2')$ und damit die Behauptung. □

In einer konkreten Implementierung des Algorithmus $\mathcal{D}$ kann die Funktion *simplify* mit dem Regelnamen parametrisiert werden, und abhängig davon einfache oder komplexere Vereinfachungen berechnen. Beispielsweise könnte man *simplify* so gestalten,

daß nur für die LET-Regel eine echte Berechnung stattfindet und sonst immer die identische Substitution geliefert wird. Dies kann man ausnutzen, um z.B. die Restriktionen in einer bestimmten Reihenfolge zu vereinfachen, um unnötigen Aufwand einzusparen.

4.7 Essentielle Typvariablen

Zusätzlich zur Anwendung vereinfachender Substitutionen auf eine Typaussage $C, \Gamma \vdash M : \tau$ besteht noch die Möglichkeit der Entfernung von Restriktionen aus der Restriktionsmenge C, die auf die Menge der möglichen Instanzierungen von τ keinen Einfluß haben. Solche Restriktionen entstehen u.a. immer dann, wenn ein Ausdruck Teilausdrücke enthält, deren Typ für den Typ des Gesamtausdrucks irrelevant ist. So hängt beispielsweise der Typ des Ausdrucks $(\lambda x.\lambda y.x)3M$ nicht vom Typ des Teilausdrucks M ab, sodaß die in C vorhandenen Restriktionen, welche die Typisierbarkeit von M garantieren, problemlos entfernt werden können. Allerdings ist dies nur erlaubt, falls die zu entfernenden Restriktionen auch erfüllbar sind.

Definition 4.41. Sei $C, \Gamma \vdash M : \tau$ eine Typaussage. $p \in C$ heilt *essentiell*, falls

$$\mathcal{I}(\varphi(\text{gen}(\Gamma, \tau | C)) \neq \mathcal{I}(\varphi(\text{gen}(\Gamma, \tau | (C - p)))). \tag{4.16}$$

für jede Abbildung $\varphi \in (\mathcal{FV}(\Gamma) \cup \mathcal{V}(\tau)) \mapsto T_F(\emptyset)$, die den freien Variablen in Γ und τ monomorphe Typausdrücke zuordnet. Die Menge der essentiellen Restriktionen einer Typaussage bezeichnen wir mit $\text{ESS}(C, \Gamma \vdash M : \tau)$.

Das Typdeduktionssystem könnte nun so erweitert werden, daß nicht essentielle Restriktionen aus Restriktionsmenge entfernt werden dürfen, vorausgesetzt sie sind erfüllbar:

$$\text{[ESS]} \quad \frac{C \cup D, \Gamma \vdash M : \tau \qquad C = \text{ESS}(C \cup D, \Gamma \vdash M : \tau) \qquad \models D}{C, \Gamma \vdash M : \tau}$$

Nun ist die Überprüfung von Gleichung 4.16 in der Regel nicht effizient durchführbar. Jedoch kann man ein hinreichendes, rein syntaktisches Kriterium definieren, das sich effizient überprüfen läßt. Dazu überlegt man sich, daß eine Restriktion $p \in D$

sicher dann essentiell ist, wenn sie eine Typvariable enthält, die in τ oder in Γ frei vorkommt, oder indirekt über andere Restriktionen in C verbunden ist.

Damit können wir folgende Deduktionsregel definieren:

$$[\text{DROP}]\quad \frac{C \cup D, \Gamma \vdash M : \tau \qquad D \cap (\mathcal{FV}(\Gamma) \cup \mathcal{V}(\tau|C)) = \emptyset \qquad \models D}{C, \Gamma \vdash M : \tau}$$

Ob eine Restriktionsmenge C einer Typisierung $C, \Gamma \vdash \tau$ eine Teilmenge D enthält, die das in Regel DROP angegeben Kriterium erfüllt, kann leicht ermittelt werden. Dazu definieren wir eine Äquivalenzrelation auf den Variablen $\mathcal{V}(\tau|C) \cup \mathcal{FV}(\Gamma)$:

Definition 4.42. Ein Typvariable heißt *beobachtbar*, falls eine Typvariable $\beta \in \mathcal{V}(\tau) \cup \mathcal{FV}(\Gamma)$ existiert, sodaß α ein Element der Äquivalenzklasse $[\beta]_{\simeq_C}$ ist, wobei $\simeq_C$ der reflexive, transitive und symmetrische Abschluß der Relation $\text{CON}(C)$ ist, die für $\alpha, \beta \in \mathcal{V}(C)$ wie folgt definiert wird:

$$(\alpha, \beta) \in \text{CON}(C) \iff \exists p \in C : \alpha \in \mathcal{V}(p), \beta \in \mathcal{V}(p)$$

Die Menge der beobachtbaren Restriktionen einer Typaussage, d.h. Restriktionen die wenigstens eine beobachtbare Typvariable enthalten, bezeichnen wir mit $\text{BEO}(C, \Gamma \vdash M : \tau)$.

Mit dieser Definition kann nun Algorithmus $\mathcal{D}[\![M]\!]\Gamma = (\tau, C)$ so abgeändert werden, daß an einer beliebigen Stelle nach der Vereinfachung von C mit *simplify*, alle Restriktionen die nicht in $B = \text{BEO}(C, \Gamma \vdash M : \tau)$ liegen aus C entfernt werden, sofern $C - B$ erfüllbar ist.. Problematisch ist jedoch die Überprüfung der Erfüllbarkeit, der wir uns im nächsten Kapitel zuwenden.

4.8 Diskussion

Mit dem hier vorgestellten System der eingeschränkten Typen, haben wir einen allgemeinen Ansatz gefunden, der es erlaubt, Typinferenzprobleme ganz unabhängig von der Art der Einschränkungen zu untersuchen. Wir haben gezeigt, daß LET-Polymorphismus unabhängig von der Art der Prädikate funktioniert, effiziente Algorithmen zur Berechnung allgemeinster Typen entwickelt und gezeigt, daß die Algorithmen sowohl wohldefiniert, als auch vollständig sind. Durch die Einführung des

Konzepts der allgemeinsten Vereinfachungen konnten wir eine kanonische Repräsentation allgemeinster Typen finden, die in jedem Prädikatsystem definiert ist. Darüber hinaus spezialisiert Algorithmus $\mathcal{D}$ zu Milners Algorithmus $\mathcal{W}$, was beweist, daß unser System eine konservative Erweiterung des Milner-System darstellt. Außerdem zeigt unsere Untersuchung, daß man die eigentliche Inferenz, also den Unifikations-, bzw. Restriktionsauflösungsaspekt, aus Algorithmus $\mathcal{W}$ herauslösen kann, und verallgemeinert somit ein bekanntes Resultat für einfach polymorphe Systeme ohne LET-Konstrukt.

Algorithmus $\mathcal{D}$ wurde auch in anderen Gebieten als der Typinferenz für Überladungen und Konversionen angewendet. Beispielsweise benutzten ihn A. Appel und Z. Shao für ihre Arbeit mit dem Titel „Smartest Recompilation“ [SA93], um minimale Modul-Schnittstellen für ML-artige Programmiersprachen zu inferieren, die garantieren, daß ein Modul nur dann neu übersetzt werden muß, wenn sich seine Implementierung ändert. Von M. Hanus et. al. wurde ein Typinferenzalgorithmus für die objektorientierte Sprache „ObjectCurry“ [HHN01] entwickelt, der auf Algorithmus $\mathcal{D}$ basiert.

Das System der eingeschränkten Typen wurde vom Autor erstmals in [Kae92] vorgestellt. Ein ähnliches, unabhängig entwickeltes System wurde von Mark Jones in [Jon92] vorgestellt und später in [Jon95] erweitert. Wir fassen kurz die wesentlichen Unterschiede zusammen:

Das in [Jon92] vorgestellte System der sogenannten qualifizierten Typen basiert auf der gleichen Grundidee wie das System der eingeschränkten Typen: Typausdrücke werden durch eine Menge von Prädikatausdrücken ergänzt. Im Unterschied zu dem hier vorgestellten generischen Typdeduktionssystem, beschränkt Jones jedoch sein Typdeduktionssystem auf das übliche Milner-System, die Deduktionsregel der Applikation lautet dort:

$$[\text{APP-Jones}]\quad \frac{C, \Gamma \vdash M_1 : \tau_a \to \tau_r \qquad C, \Gamma \vdash M_2 : \tau_a}{C, \Gamma \vdash M_1\, M_2 : \tau_r}$$

Konsequenterweise verwendet der Typinferenzalgorithmus den üblichen Unifikationsalgorithmus und entspricht somit weitgehend dem Algorithmus $\mathcal{W}$. Daher kann auch keine Trennung zwischen Restriktionssammlung und anschließender Restriktionsvereinfachung bzw. Auflösung erfolgen.

Die Frage der Erfüllbarkeit von Restriktionen wird erst in [Jon95] aufgegriffen. Dort gilt eine Restriktionsmenge als erfüllbar, wenn sie die logische Konsequenz einer gewissen Grundmenge von Restriktionen P_0 ist, d.h. falls $P_0 \Vdash C$ gilt, wobei in der Regel $P_0 = \emptyset$. Dadurch wird die Einführung einer Interpretationsfunktion für Prädikate vermieden. Auch die Frage nach der Vereinfachung von Typaussagen, bzw. Restriktionsmengen wird erst in [Jon95] untersucht. Dazu werden zwei unterschiedliche Konzepte definiert: Vereinfachungen und Verbesserungen von Restriktionsmengen. Verbesserungen entsprechen unseren Vereinfachungen, Vereinfachungen entsprechen dem Austausch von Restriktionsmengen durch andere Mengen, welche die gleiche Lösungsmenge besitzen. Dabei dürfen Prädikate beliebig ausgetauscht werden, während im hier vorgestellten System nur der Austausch durch logisch äquivalenten Mengen gemäß der Implikationsrelation erlaubt ist.

Ein weiterer Unterschied besteht in der Behandlung lokaler Definitionen der Form **let** $x = M_1$ **in** M_2. Im System der qualifizierten Typen werden alle Restriktionen, die für die Ableitung des Typs für M_1 benötigt werden, in den generischen Typ für den Bezeichner x zur Ableitung des Typs für M_2 übernommen. Die LET-Regel lautet damit:

$$\text{[LET-Jones]} \quad \frac{C_1, \Gamma \vdash M_1 : \tau \qquad C_2, \Gamma + [x : \text{gen}'(\Gamma, \tau_1 | C_1)] \vdash M_2 : \tau_2}{C_2, \Gamma \vdash \textbf{let } x = M_1 \textbf{ in } M_2 : \tau_2}$$

wobei die Verallgemeinerung im Kontext von Typannahmen wie folgt definiert wird: $\text{gen}'(\Gamma, \tau | C) = \forall \overline{\alpha_n}.\tau | C$ mit $\{\overline{\alpha_n}\} = \mathcal{V}(\tau | C) - \mathcal{FV}(\Gamma)$. Damit sind natürlich dann Ausdrücke typisierbar, die untypisierbare Teilausdrücke enthalten, was wir in unserem System ausgeschlossen haben. Da in der ersten Veröffentlichung der Erfüllbarkeitsbegriff für Restriktionen fehlt, ist dies nicht weiter tragisch. Jones umgeht das Problem in der Folgeveröffentlichung, indem er Ausdrücke mit nicht verwendeten LET-Definitionen einfach verbietet.[22]

Beide Ansätze zur Behandlung des LET-Konstrukts haben ihre Vor- und Nachteile: die Jones-Regel ist besser geeignet als Modell für sogenannte „Open-World“-Systeme, in denen die Menge der gültigen Instanzen für Restriktionen nicht feststeht, sondern inkrementell vom Anwender durch neue Instanzdefinitionen erweitert werden kann. In einem solchen System können bei der Typisierung von Bibliotheksfunktionen nur

[22] Ein Trick, der gerne angewendet wird, z.B. auch in [Hen96].

offensichtliche Wiedersprüche aufgedeckt werden. In einem „Closed World“-Ansatz, wie z.B. in SAMPλE, ist die Menge der möglichen Instanzen für Restriktionen schon an der Definitionsstelle der Funktion bekannt, und daher kann die Deklaration sofort auf die Existenz einer gültigen Instanzierung überprüft werden.

Kapitel 5

Entscheidbare Prädikatsysteme

Wir wenden uns nun der Frage zu, für welche Klassen von Prädikaten die Frage nach der Existenz von erfüllenden Substitutionen bzw. Lösungen sowie genauesten Vereinfachungen positiv beantwortet werden kann. Wir werden zeigen, daß strukturelle Konversionen und parametrische Überladungen sowie nicht rekursiv definierte Prädikate integriert behandelt werden können. Dazu müssen wir zunächst einen Formalismus zur Spezifikation der Prädikate wählen. Im Grunde ist die Wahl des Formalismus von untergeordneter Bedeutung, solange er es erlaubt, interessante Klassen von Prädikaten zu untersuchen. In dieser Arbeit werden wir Ersetzungregeln für Restriktionsmengen verwenden, denkbar wäre aber auch eine geeignete Einschränkung des Prädikatenkalküls erster Ordnung. Es ist klar daß man ein z.B. ein Logikprogramm in PROLOG schreiben könnte, daß alle erfüllenden Substitutionen bzw. Lösungen aufzählt. Jedoch lassen sich auf diese Weise genaueste Vereinfachungen nicht berechnen.[1]

Wir geben zunächst in Abschnitt 5.1 eine Definition der Restriktionsersetzungsysteme an und betrachten ein Beispielsystem, dessen Handhabbarkeit in den folgenden Abschnitten bewiesen wird. Wir betrachten zunächst in Abschnitt 5.2 nicht rekursiv definierte Prädikate, die insbesondere für endliche Überladungen relevant sind und zeigen dann in Abschnitt 5.3, daß ein Restriktionsproblem über *strukturelle Äquivalenz* bzw. *strukturelle Ähnlichkeit* erzwingenden Prädikaten, in eine endliche Menge *atomar gelöster* Restriktionsmengen transformiert werden kann. Ist keines dieser Probleme lösbar, so ist das Originalproblem unlösbar. Ansonsten existiert eine Lösung und auch eine genaueste Vereinfachung des Originalproblems. Rekursive Prädikate und endliche Überladungen können in einem einzigen System zusammengefaßt werden. Anschließend zeigen wir in Abschnitt 5.4, daß die Lösbarkeit von Restriktionsmengen über *zerlegbaren* Prädikaten entscheidbar ist und eine genaueste

[1] Siehe dazu auch [FSVY91].

Vereinfachung berechnet werden kann. Eine Diskussion der Ergebnisse beschließt das Kapitel.

5.1 Restriktionsersetzungsysteme

Wir setzen im Folgenden die Existenz einer Ersetzungsrelation $\longrightarrow\!\!\!\!\!\rightarrow$ auf Restriktionsmengen voraus, welche die Lösbarkeit von Restriktionsmengen definiert. Wir legen fest:

$$\hat{p}(\overline{t_n}) \stackrel{\text{def}}{=} \{p(\overline{t_n})\} \longrightarrow\!\!\!\!\!\rightarrow^* \emptyset \tag{5.1}$$

Daraus folgt sofort:

$$L \models C \iff L(C) \longrightarrow\!\!\!\!\!\rightarrow^* \emptyset \tag{5.2}$$

Dabei bezeichnet $\longrightarrow\!\!\!\!\!\rightarrow^*$ den reflexiven und transitiven Abschluß von $\longrightarrow\!\!\!\!\!\rightarrow$. Eine natürliche Forderung an $\longrightarrow\!\!\!\!\!\rightarrow^*$ ist, daß $\longrightarrow\!\!\!\!\!\rightarrow^*$ sowohl konfluent als auch terminierend ist. Außerdem fordern wir, daß durch $\longrightarrow\!\!\!\!\!\rightarrow$ keine neuen Variablen eingeführt werden: $C \longrightarrow\!\!\!\!\!\rightarrow D \Rightarrow \mathcal{V}(D) \subseteq \mathcal{V}(C)$. Implikation von Restriktionsmengen kann dann leicht durch die Gleichung

$$C \Vdash D \iff \exists C' \subseteq C.C' \leftarrow\!\!\!\!\!\rightarrow^* D \tag{5.3}$$

festgelegt werden. Dabei bezeichnet $\leftarrow\!\!\!\!\!\rightarrow^*$ die reflexive, transitive und symmetrische Hülle der Relation $\longrightarrow\!\!\!\!\!\rightarrow$. Aus dieser Definition folgt sofort, daß die Implikation und Äquivalenz von Restriktionsmengen aufgrund der Konfluenz und Termination von $\longrightarrow\!\!\!\!\!\rightarrow$ entscheidbar ist. Außerdem ist Implikation abgeschlossen unter Substitutionen, aufgrund der Abgeschlossenheit von Termersetzungssystemen unter Substitutionen.

Proposition 5.1. *Die durch Gleichung (5.3) definierte Relation $\Vdash$ ist eine Implikationsrelation.*

Im Folgenden nehmen wir an, daß die Relation $\longrightarrow\!\!\!\!\!\rightarrow$ durch eine endliche Menge von Regeln R der Form $l \longrightarrow\!\!\!\!\!\rightarrow r$ spezifiziert wird, wobei l und r Restriktionsmengen sind und $\mathcal{V}(r) \subseteq \mathcal{V}(l)$ gilt. Die Ersetzungsrelation $\longrightarrow\!\!\!\!\!\rightarrow$ wird dann festgelegt als die Menge von Paaren $(C \cup D, C \cup S(r))$, sodaß $D = S(l)$ für ein $l \longrightarrow\!\!\!\!\!\rightarrow r \in R$. Man sieht sofort, daß die so definierte Relation $\longrightarrow\!\!\!\!\!\rightarrow$ unter Substitutionen abgeschlossen ist: $A \longrightarrow\!\!\!\!\!\rightarrow B \Rightarrow S(A) \longrightarrow\!\!\!\!\!\rightarrow S(B)$.

$$
\begin{array}{ll}
int \lhd int \longrightarrow\!\!\!\!\!\rightarrow \emptyset & p_{\leq}(int) \longrightarrow\!\!\!\!\!\rightarrow \emptyset \\
real \lhd real \longrightarrow\!\!\!\!\!\rightarrow \emptyset & p_{\leq}(real) \longrightarrow\!\!\!\!\!\rightarrow \emptyset \\
int \lhd real \longrightarrow\!\!\!\!\!\rightarrow \emptyset & p_{\leq}(a \times b) \longrightarrow\!\!\!\!\!\rightarrow p_{\leq}(a), p_{=}(a), p_{\leq}(b) \\
a \to b \lhd c \to d \longrightarrow\!\!\!\!\!\rightarrow c \lhd a, b \lhd d & p_{\leq}(list(a)) \longrightarrow\!\!\!\!\!\rightarrow p_{\leq}(a), p_{=}(a) \\
a \times b \lhd c \times d \longrightarrow\!\!\!\!\!\rightarrow a \lhd c, b \lhd d & \\
list(a) \lhd list(b) \longrightarrow\!\!\!\!\!\rightarrow a \lhd b & p_{\in}(a, list(a)) \longrightarrow\!\!\!\!\!\rightarrow p_{=}(a) \\
set(a) \lhd set(b) \longrightarrow\!\!\!\!\!\rightarrow a \lhd b & p_{\in}(a, set(a)) \longrightarrow\!\!\!\!\!\rightarrow p_{=}(a) \\
ref(a) \lhd ref(b) \longrightarrow\!\!\!\!\!\rightarrow a = b & p_{\in}(b, array(a, b)) \longrightarrow\!\!\!\!\!\rightarrow p_{=}(b) \\
 & \\
p_{=}(int) \longrightarrow\!\!\!\!\!\rightarrow \emptyset & p_{\downarrow}(list(a), int, a) \longrightarrow\!\!\!\!\!\rightarrow \emptyset \\
p_{=}(real) \longrightarrow\!\!\!\!\!\rightarrow \emptyset & p_{\downarrow}(array(a, b), a, b) \longrightarrow\!\!\!\!\!\rightarrow p_{=}(a) \\
p_{=}(a \times b) \longrightarrow\!\!\!\!\!\rightarrow p_{=}(a), p_{=}(b) & p_{\downarrow}(a \to b, a, b) \longrightarrow\!\!\!\!\!\rightarrow \emptyset \\
p_{=}(list(a)) \longrightarrow\!\!\!\!\!\rightarrow p_{=}(a) & \\
p_{=}(ref(a)) \longrightarrow\!\!\!\!\!\rightarrow \emptyset & p_{fst}(\alpha \times \beta, \alpha) \longrightarrow\!\!\!\!\!\rightarrow \emptyset \\
p_{=}(set(a)) \longrightarrow\!\!\!\!\!\rightarrow p_{=}(a) & p_{fst}(\alpha \times \beta \times \gamma, \alpha) \longrightarrow\!\!\!\!\!\rightarrow \emptyset
\end{array}
$$

Abbildung 5.1: Ein Ersetzungssystem für strukturelle Konversionen und überladene Operatoren

Abbildung 5.1 enthält beispielhaft die Spezifikation eines nicht trivialen Systems überladener Operatoren sowie einer Konvertierbarkeitsrelation $\lhd$. Im Einzelnen wird folgendes festgelegt:

Ganze bzw. reelle Zahlen können in sich selbst und in reelle Zahlen konvertiert werden. Konvertierbarkeit von Funktionen ist monoton im Funktionsresultat und antimonoton im Funktionsargument. Die Konversion von Paaren wird monoton auf die Konversion ihrer Elemente zurückgeführt, Listen vom Typ a können in Listen vom Typ b konvertiert werden, falls a nach b konvertiert werden kann. Referenzzellen dagegen lassen sich überhaupt nicht konvertieren.

Das Prädikat $p_{=}$ legt die zulässigen Argumente für den überladenen Operator $=$ vom Typ $\forall\alpha.\alpha \to \alpha \to bool | p_{=}(\alpha)$ fest: Zahlen können verglichen werden, die Listengleichheit und die Mengengleichheit wird auf die Gleichheit der Elemente zurückgeführt und Referenzzellen können unabhängig von den in ihnen gespeicherten Werten ver-

glichen werden.

Die Regeln für den überladen Operator $\leq$: $\forall\alpha.\alpha \to \alpha \to bool|p_{\leq}(\alpha)$ legen fest, auf welchen Daten die Ordnungsfunktion $\leq$ definiert ist. Es sind dies: Zahlen, Listen und Paare. Für Listen soll $\leq$ die lexikographische Ordnung sein, daher ist $\leq$ nur zulässig, falls auf den Elementen sowohl $\leq$ als auch $=$ definiert ist. Ähnliches gilt für Paare.

Das Prädikat $p_{\in}$ spezifiziert die zulässigen Argumenttypen für einen überladenen $\in$-Operator vom Typ $\forall\alpha,\beta.\alpha \to \beta|p_{\in}(\alpha,\beta)$: Werte vom Typ a können in Listen vom Typ $list(a)$ gesucht werden, falls a als Argument des Gleichheitsoperators zulässig ist, entsprechendes gilt für Mengen. Für Felder mit Indextyp a und Elementtyp b ist das Suchen nach Werten vom Typ b erlaubt.

Das Prädikat $p_{\downarrow}$ beschreibt die zulässigen Argumente eines überladenen Selektionsoperators $\downarrow$ vom Typ $\forall\alpha,\beta,\gamma.\alpha \to \beta \to \gamma.\alpha \to \beta \to \gamma$: erlaubt werden Listen- und Feldindexierung sowie die Applikation von Funktionen.[2]

Schließlich beschreibt p_{fst} die zulässigen Argumente eines überladenen Operators fst : $\forall\alpha,\beta.\alpha \to \beta|p_{fst}(\alpha,\beta)$ zur Selektion der ersten Komponente in n-Tupeln.

Die Operatoren $=$ und $\leq$ besitzen eine Menge von Typinstanzen, die schon im System der parametrischen Überladungen darstellbar war. Die Ersetzungsregeln für parametrische Überladungen können dabei auf systematische Weise aus einer Überladungsannahme konstruiert werden: gegeben eine Überladungsannahme O, definiert man für jeden überladenen Operator x mit $O(x) = \langle\omega, s\rangle$ ein Prädikatsymbol p_x mit $\rho(p_x) = 1$ und für jeden Ausdruck $f(\overline{\alpha_n}) \in s$ eine Ersetzungsregel der Form $p_x(f(\overline{a_n})) \longrightarrow\!\!\!\!\!\rightarrow \widehat{\alpha_1} \cup \cdots \cup \widehat{\alpha_n}$, wobei $\widehat{\alpha_X} = \bigcup_{i=1..n}\{p_y(a_i) \mid y \in X\}$ und die a_i neue unsortierte Typvariablen sind.

Wendet man diesen Algorithmus auf die Überladungsannahme

$$= \mapsto \langle \$ \to \$ \to bool, \{int, real, \alpha_{\{=\}} \times \beta_{\{=\}}\}, list(\alpha_{\{=\}}), ref(\alpha), set(\alpha_{\{=\}})\rangle,$$
$$\leq \mapsto \langle \$ \to \$ \to bool, \{int, real, \alpha_{\{=,\leq\}} \times \beta_{\{\leq\}}\}, list(\alpha_{\{=,\leq\}})\rangle$$

an, so erhält man die in Abbildung 5.1 angegebenen Regeln für $p_{=}$ und $p_{\leq}$.

Die Typisierung der Operatoren $\in$, $\downarrow$ und fst ist im System der parametrischen Überladungen, wegen der Einschränkung auf eine einzige Überladungsvariable, nicht

[2] $\downarrow$ könnte auch zur Selektion in n-Tupeln erweitert werden.

möglich.

Wir werden im Folgenden zeigen, daß die Restriktionsauflösung für dieses System von Prädikaten entscheidbar ist. Dazu definieren wir Prädikatklassen, die das obige System als Spezialfall beinhalten.

5.2 Nichtrekursive Prädikate

Die einfachste Klasse von Restriktionsmengenersetzungssystemen mit lösbaren Restriktionsproblemen sind diejenigen, die dem „traditionellen" Überladungsansatz entsprechen, bei dem zusätzlich zur syntaktischen Gleichheit von Termen nur eine endliche Menge von Überladungsinstanzen vorhanden ist. Typisches Beispiel hierfür ist die Programmiersprache ADA [als83] [3], sowie HOPE und die arithmetischen Operatoren von Standard-ML.[4] Der Lösungsalgorithmus für diese Klasse von Prädikatsystemen ist denkbar einfach: Berechne den allgemeinsten Unifikator $U(C)$ aller Restriktionen der Form $\tau_1 = \tau_2$ der gegebenen Restriktionsmenge und zähle eine Menge minimaler Substitutionen S auf, sodaß $S(U(C))$ in die leere Restriktionsmenge überführt werden kann. Aufgrund von Lemma 4.36 kann man dann sogar die genaueste Vereinfachung der Restriktionsmenge C durch Verknüpfung von U und dem Supremum aller berechneten Substitutionen erhalten.

Zur Aufzählung einer Menge minimaler Substitutionen definieren wir eine Transformationsrelation $\Longrightarrow_1$ auf Restriktionsmengen, sodaß jede Normalform von C unter $\Longrightarrow_1$ entweder unlösbar ist oder aber in gelöster Form, die dann eine Lösung von C bestimmt. Diese Vorgehensweise orientiert sich stark an dem in [MM82] angegebenen Unifikationsalgorithmus für Terme freier Algebren, unsere Notation ähnelt jedoch mehr [Nip].

Definition 5.2. Sei $C = \{s_1 = t_1, \ldots, s_n = t_n\} \cup \{p_1(\overline{u}_1), \ldots, p_l(\overline{u}_l)\}$ ein Restriktionsproblem. C ist in *atomar gelöster Form*, falls

1. $s_i \in \mathcal{V}$ für alle i
2. $s_i \neq s_j \wedge s_i \notin \mathcal{V}(t_j)$ für alle $i \neq j$

[3] ADA is a registered trademark of the U. S. Government, Ada Joint Program Office.
[4] Siehe dazu auch [Hud89].

3. $s_i \notin \mathcal{V}(u_{jk})$ für alle i, j, k

4. $u_{jk} \in F_0 \cup \mathcal{V}$ für alle j, k

5. $\overline{u}_i \cap \mathcal{V} \neq \emptyset$ für alle i

Falls C keinen Prädikatterm enthält, also $l = 0$ gilt, dann sagen wir C ist in *gelöster Form*.

Man beachte, daß wir Restriktionen der Form $a = b$ als ein ungeordnetes Paar von Termen $\{a, b\}$ auffassen, sodaß ein Restriktionsproblem atomar gelöst ist, wenn die Gleichheitsrestriktionen so sortiert werden können, daß die Bedingungen der obigen Definition erfüllt werden.

Lemma 5.3. *Falls C in atomar gelöster Form vorliegt, dann ist die Substitution $\vec{C} = \{s_1 \mapsto t_1, \ldots, s_n \mapsto t_n\}$ idempotent und $\vec{C}(C)$ besitzt die gleiche Lösungsmenge wie $\{p_1(\overline{u}_1), \ldots, p_l(\overline{u}_l)\}$.*

Beweis. Idempotenz folgt sofort aus Bedingung 1 und 2. Da $\vec{C}$ keine Variablen involviert, die als Argumente von Restriktionen $p(\overline{u})$ vorkommen, gilt $\vec{C}(C) = \{t_1 = t_1, \ldots, t_n = t_n\} \cup \{p_1(\overline{u}_1), \ldots, p_l(\overline{u}_l)\}$. Offensichtlich wird eine Gleichung der Form $t = t$ von jeder Substitution erfüllt, somit gilt $\mathcal{L}(\vec{C}(C)) = \mathcal{L}(\{p_1(\overline{u}_1), \ldots, p_l(\overline{u}_l)\})$. □

Wir gehen im Folgenden davon aus, daß die linke Seite von Ersetzungsregeln aus einem einzigen Prädikatterm besteht, der als Kopf das Prädikatsymbol besitzt, und daß die linken Seiten der Regeln paarweise nicht unifizierbar sind. Falls kein Prädikat rekursiv ist, d.h. falls für keine Restriktion $p(\overline{s_n})$ eine Menge D existiert, sodaß $p(\overline{s_n}) \longrightarrow\!\!\!\!\rightarrow^+ D$ und $p(\overline{t_n}) \in D$ gilt, terminiert die in Abbildung 5.2 angegebene Transformationsrelation und liefert eine vollständige Menge minimaler Lösungen.

Definition 5.4. Eine Transformationsrelation $\Longrightarrow$ auf Restriktionsmengen heißt

1. wohldefiniert, falls $C \Longrightarrow C' \Rightarrow \mathcal{L}(C') \subseteq \mathcal{L}(C)$,

2. (atomar) vollständig, falls $\mathcal{L}(C)_{|\mathcal{V}(C)} \subseteq \bigcup_{C \Longrightarrow C'} \mathcal{L}(C')_{|\mathcal{V}(C)}$ für alle Restriktionsmengen C, die nicht in (atomar) gelöster Form vorliegen.

Löschung	$C \cup \{\tau = \tau\}$	$\Longrightarrow_1$	C
Zerlegung	$C \cup \{f(\overline{\tau_n}) = f(\overline{\tau'_n})\}$	$\Longrightarrow_1$	$C \cup \{\tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n\}$
Substitution	$C \cup \{\alpha = \tau\}$	$\Longrightarrow_1$	$\{\alpha \mapsto \tau\}(C) \cup \{\alpha = \tau\}$ falls $\alpha \in \mathcal{V}(C) - \mathcal{V}(\tau),\ \tau \in \mathcal{V} \Rightarrow \tau \in \mathcal{V}(C)$
Anpassung	$C \cup \{p(\overline{\tau_n})\}$	$\Longrightarrow_1$	$C \cup \overline{r_m} \cup \{\tau_1 = l_1, \ldots, \tau_n = l_n\}$ falls $p(\overline{l_n}) \longrightarrow\!\!\!\!\!\rightarrow \overline{r_m}$ eine Ersetzungsregel ist, entfernt von $\mathcal{V}(C) \cup \mathcal{V}(\overline{\tau_n})$

Abbildung 5.2: Transformation zur Lösung nichtrekursiver Restriktionsprobleme

Die Einschränkung des Definitionsbereiches von Lösungen in der Definition der Vollständigkeit ist notwendig, da Transformationen in der Regel neue Variablen einführen.

Satz 5.5. *$\Longrightarrow_1$ ist wohldefiniert und vollständig. Falls kein Prädikat rekursiv ist, dann terminiert $\Longrightarrow_1$. Jede Normalform unter $\Longrightarrow_1$ ist entweder unlösbar oder in gelöster Form.*

Beweis. Löschung, Zerlegung und Substitution sind die üblichen Regeln zur syntaktischen Unifikation, Anpassung beschreibt die Anwendung von Ersetzungsregeln. Die Regelnamen werden im Folgenden mit ihren Anfangsbuchstaben abgekürzt.

Wohldefiniertheit. Wir zeigen durch Fallunterscheidung über die angewendete Transformationsregel, daß für $C \Longrightarrow_1 C'$ und $S \in \mathcal{L}(C')$ auch $S \in \mathcal{L}(C)$ gilt.

L: Sei $S \in \mathcal{L}(C)$. Da $S'(\tau) = S'(\tau)$ für jede Substitution S', gilt trivialerweise $S \in \mathcal{L}(C \cup \{\tau = \tau\})$.

Z: Sei $S \in \mathcal{L}(C \cup \{\tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n\})$. Dann gilt $S(\tau_i) = S(\tau'_i)$ für alle i und daher $S(f(\overline{\tau_n}) = S(f(\overline{\tau'_n}))$ und somit $S \in \mathcal{L}(C \cup \{f(\overline{\tau_n}) = f(\overline{\tau'_n})\})$

S: Sei $S \in \mathcal{L}(\{\alpha \mapsto \tau\}(C) \cup \{\alpha = \tau\})$. Dann gilt offensichtlich $S(\alpha) = S(\tau)$ sowie $S \circ \{\alpha \mapsto \tau\} = S$. Dies impliziert $S(C) = S(\{\alpha \mapsto \tau\}(C))$ und daher gilt

$S \in \mathcal{L}(C \cup \{\alpha = \tau\})$.

A: Sei $S \in \mathcal{L}(C \cup \overline{r_m} \cup \{\tau_1 = l_1, \ldots, \tau_n = l_n\})$ für eine literale Kopie einer Ersetzungsregel $p(\overline{l_n}) \longrightarrow\!\!\!\!\rightarrow \overline{r_m} \in R$. Zu zeigen ist $S(\{p(\overline{\tau_n})\}) \longrightarrow\!\!\!\!\rightarrow^* \emptyset$. Aus $S(\tau_i) = S(l_i)$ für alle i folgt $S(p(\overline{\tau_n})) \longrightarrow\!\!\!\!\rightarrow S(\overline{r_m})$, da $\longrightarrow\!\!\!\!\rightarrow$ abgeschlossen ist unter Substitutionen. Wegen $S(\overline{r_m}) \longrightarrow\!\!\!\!\rightarrow^* \emptyset$ gilt aber auch $S(\{p(\overline{\tau_n})\}) \longrightarrow\!\!\!\!\rightarrow^* \emptyset$ und somit $S \in \mathcal{L}(C \cup \{p(\overline{\tau_n})\})$.

Vollständigkeit. Zu zeigen ist, daß für jede Lösung S eines Restriktionsproblems C, das nicht in gelöster Form vorliegt, eine Transformationsregel $\overset{x}{\Longrightarrow}$ existiert, sodaß $C \overset{x}{\Longrightarrow} C'$ und $S_{|\mathcal{V}(C)} \in \mathcal{L}(C')$. O.B.d.A. gelte $dom(S) = \mathcal{V}(C)$. Der Beweis erfolgt durch Fallunterscheidung über die Form von C.

L: $S \models C \cup \{\tau = \tau\} \Rightarrow S \models C$

Z: $S \models C \cup \{f(\overline{\tau_n}) = f(\overline{\tau'_n})\} \Rightarrow S \models C \cup \{\tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n\}$

S: $S \models C \cup \{\alpha = \tau\} \Rightarrow S \models \{\alpha \mapsto \tau\}(C) \cup \{\alpha = \tau\}$

A: Sei $C_0 = C \cup \{p(\overline{\tau_n})\}$ und es gelte $S \models C_0$. Dann muß es aufgrund der Definition der Erfüllbarkeit von Restriktionsmengen eine literale Kopie einer Ersetzungsregel $p(\overline{l_n}) \longrightarrow\!\!\!\!\rightarrow \overline{r_m}$ mit $\mathcal{V}(\overline{l_n}) \cap \mathcal{V}(C_0) = \emptyset$ und eine Substitution S' mit $dom(S') = \mathcal{V}(\overline{l_n})$ geben, sodaß $S'(\overline{l_n}) = S(\overline{\tau_n})$ und $S(C) \cup S'(\overline{l_n}) \longrightarrow\!\!\!\!\rightarrow S(C) \cup S'(\overline{r_m}) \longrightarrow\!\!\!\!\rightarrow^* \emptyset$. Sei $S'' = S \circ S'$, dann gilt $S''(l_i) = S''(\tau_i)$, $S''(C) \longrightarrow\!\!\!\!\rightarrow^* \emptyset$, $S''(\overline{r_m}) \longrightarrow\!\!\!\!\rightarrow^* \emptyset$, $S''\{\tau_1 = l_1, \ldots, \tau_n = l_n\} \longrightarrow\!\!\!\!\rightarrow^* \emptyset$ und daher $S'' \in \mathcal{L}(C \cup \overline{r_m} \cup \{\tau_1 = l_1, \ldots, \tau_n = l_n\})$. Aus $S = S''_{|\mathcal{V}(C_0)}$ folgt die Behauptung.

Termination. Da keines der Prädikate rekursiv definiert ist, gibt es für jede Transformationssequenz eine obere Schranke für die Anzahl der Anwendungen der Regel A, und zwar unabhängig von der Größe der Argumente von Prädikattermen in C. Sei $B(C)$ diese Schranke, $U(C)$ die Anzahl der Variablen, die nicht nur einmal auf der linken Seite einer Gleichung der Form $\alpha = \tau$ vorkommen und $S(C)$ die Anzahl der in C vorkommenden Symbole. Sei $F(C)$ definiert als das Tripel $\langle B(C), U(C), S(C)\rangle$. Die Termination folgt durch noethersche Induktion aus der Tatsache, daß für alle

Restriktionsmengen $C \Longrightarrow_1 C'$ die Ungleichung $F(C') < F(C)$ erfüllt ist. Dabei ist $<$ die übliche Striktordnung auf Tripeln natürlicher Zahlen

$$(n_1, n_2, n_3) < (n_1', n_2', n_3') \iff \begin{cases} n_1 < n_1' & \text{oder} \\ n_1 = n_1' \wedge n_2 < n_2' & \text{oder} \\ n_1 = n_1' \wedge n_2 = n_2' \wedge n_3 < n_3' \end{cases}$$

L: Es gilt $S(C') < S(C)$, $U(C)$ und $B(C)$ bleiben unverändert.

Z: Es gilt $S(C') < S(C)$ und $U(C') \leq U(C)$, da einige der τ_i bzw. τ_i' in C' gelöst sein könnten. $B(C)$ bleibt unverändert.

S: $U(C') < U(C)$. $B(C)$ bleibt unverändert.

A: $B(C') = B(C) - 1$.

Der dritte Teil des Satzes folgt durch genaue Betrachtung der Normalformen unter $\Longrightarrow_1$: es ist offensichtlich, daß eine solche Normalform weder Restriktionen $p(\overline{\tau_n})$ noch Gleichungen der Form $\tau = \tau$ bzw. $f(\overline{\tau_n}) = f(\overline{\tau_n'})$ enthält, denn sonst könnte noch eine der Regeln A, L oder Z angewendet werden. Falls C eine Gleichung der Form $f(\overline{\tau_n}) = g(\overline{\tau_m'})$ enthält, dann gibt es keine Lösung für C. Für Gleichungen der Form $\alpha = \tau$ unterscheidet man zwei Fälle: Ist τ eine Variable, dann gilt entweder $\alpha \notin \mathcal{V}(C)$ oder $\tau \notin \mathcal{V}(C)$, sodaß entweder α gelöst in $C \cup \{\alpha = \tau\}$ oder τ gelöst in $C \cup \{\tau = \alpha\}$. Ist τ keine Variable und $\alpha \in \mathcal{V}(\tau)$, dann hat $\alpha = \tau$ und somit $C \cup \{\alpha = \tau\}$ keine Lösung. Gilt $\alpha \notin \mathcal{V}(\tau)$, dann kommt α nicht mehr in C vor und ist somit gelöst in $C \cup \{\alpha = \tau\}$. □

5.3 Rekursive Prädikate

Konversionen oder parametrische Überladungen, wie sie etwa zur Definition des überladenen Gleichheitsoperators benötigt werden, verlangen jedoch, daß Prädikate rekursiv definiert werden können. Dies impliziert natürlich, daß eine gegebene Restriktionsmenge im allgemeinen unendlich viele Lösungen besitzen kann. In [Mit84, FM88] wurde gezeigt, daß man für den Spezialfall der „strukturellen Konversionen“ Algorithmen angeben kann, die eine gegebene Restriktionsmenge in eine äquivalente

Restriktionsmenge umformen kann, die nur noch Restriktionen mit Basistypen und Typvariablen als Argumente enthält und daß diese sogenannten „atomaren“ Restriktionsmengen genau dann eine Lösung besitzen, falls sie eine Lösung besitzen, die alle Typvariablen durch Basistypen ersetzt.

Zwei Eigenschaften der strukturellen Konversionen sind essentiell für die Herleitung dieses Resultats: daß Konvertierbarkeit induktiv definiert ist und strukturelle Äquivalenz der zu konvertierenden Typen erzwingt. Zwei Terme heißen strukturell äquivalent ($\tau \overset{\text{se}}{\sim} \tau'$), falls sie nach Identifikation von nullstelligen Funktionssymbolen und Typvariablen gleich sind. Die Anforderung, daß die Prädikate strukturelle Gleichheit erzwingen, kann noch zur strukturellen Ähnlichkeit verallgemeinert werden. Zwei Terme heißen strukturell ähnlich ($\tau \overset{\text{ss}}{\sim} \tau'$), falls sie das gleiche Baumgerüst besitzen. Strukturelle Ähnlichkeit ermöglicht z.B. die Definition von Konversionsregeln der Form

$$list(a) \triangleleft set(b) \longrightarrow\!\!\!\!\rightarrow a \triangleleft b, p_{=}(b) \ .$$

Diese Regel erlaubt die Konversion von Listen des Typs a in Mengen vom Typ b, vorausgesetzt a ist ein Subtyp von b und b erlaubt Gleichheit.

Definition 5.6. Strukturelle Äquivalenz ($\overset{\text{se}}{\sim}$), Strukturelle Ähnlichkeit ($\overset{\text{ss}}{\sim}$)

$$\begin{aligned}
\tau \overset{\text{se}}{\sim} \tau' \iff\ & \tau, \tau' \in F_0 \cup \mathcal{V} \vee \\
& \tau = f(\tau_1, \ldots, \tau_n),\ \tau' = f(\tau'_1, \ldots, \tau'_n) \wedge \forall i = 1..n : \tau_i \overset{\text{se}}{\sim} \tau'_i \\
\tau \overset{\text{ss}}{\sim} \tau' \iff\ & \tau, \tau' \in F_0 \cup \mathcal{V} \vee \\
& \tau = f(\tau_1, \ldots, \tau_n),\ \tau' = g(\tau'_1, \ldots, \tau'_n) \wedge \forall i = 1..n : \tau_i \overset{\text{ss}}{\sim} \tau'_i
\end{aligned}$$

Korrollar 5.7. *$\overset{se}{\sim}$ und $\overset{ss}{\sim}$ sind Äquivalenzrelationen. Es gilt:*

$$\begin{aligned}
&(1)\ \tau_1 = \tau_2 \Rightarrow \tau_1 \overset{se}{\sim} \tau_2 \\
&(2)\ \tau_1 \overset{se}{\sim} \tau_2 \Rightarrow \tau_1 \overset{ss}{\sim} \tau_2
\end{aligned}$$

Beweis. Folgt sofort aus der Definition von $\overset{\text{se}}{\sim}$ und $\overset{\text{ss}}{\sim}$. □

Definition 5.8. Ein Prädikat $p \in P_n$ ist induktiv definiert, falls für alle $(f_1, \ldots, f_n) \in F^n_+$, entweder p für keinen Argumentvektor der Form $f_1(\overline{t}_1), \ldots, f_n(\overline{t}_n)$ definiert ist, oder eine Indexmenge $I(p, f_1, \ldots, f_n)$ existiert, sodaß für alle $\overline{t}_1, \ldots, \overline{t}_n \in T^m_F$:

$$p(f_1(\overline{t}_1), \ldots, f_n(\overline{t}_n)) \longrightarrow\!\!\!\!\rightarrow \bigcup_{i \in I(p, f_1, \ldots, f_n)} p_i(\overline{s_i})$$

wobei $s_{ij} \in \{t_{11}, \ldots, t_{nm}\}$.

Definition 5.9. Ein Prädikat $p \in P_n$ erzwingt strukturelle Gleichheit bzw. Ähnlichkeit, falls für alle $\overline{t_n} \in T_F^n$

$$p(t_1, \ldots, t_n) \longrightarrow\!\!\!\!\!\rightarrow^* \emptyset \Rightarrow t_1 \overset{\text{se}}{\sim} t_2 \overset{\text{se}}{\sim} \cdots \overset{\text{se}}{\sim} t_n \ .$$

bzw.

$$p(t_1, \ldots, t_n) \longrightarrow\!\!\!\!\!\rightarrow^* \emptyset \Rightarrow t_1 \overset{\text{ss}}{\sim} t_2 \overset{\text{ss}}{\sim} \cdots \overset{\text{ss}}{\sim} t_n \ .$$

Man beachte, daß strukturelle Ähnlichkeit selbst ein induktiv definierbares, strukturelle Ähnlichkeit erzwingendes Prädikat ist, wohingegen strukturelle Gleichheit und syntaktische Gleichheit sogar strukturelle Gleichheit erzwingen.

Satz 5.10. *Sei C ein Restriktionsproblem über strukturelle Ähnlichkeit erzwingenden, induktiv definierten Prädikaten. Dann existiert eine vollständige, endliche Menge $\mu SS(C)$ strukturelle Ähnlichkeit erzwingender, minimaler Substitutionen, mit der Eigenschaft:*

- $\forall S \in \mu SS(C) : p(\tau_1, \ldots, \tau_n) \in S(C) \Rightarrow \tau_1 \overset{ss}{\sim} \tau_2 \overset{ss}{\sim} \cdots \overset{ss}{\sim} \tau_n$
- $L \models C \Rightarrow \exists S \in \mu SS(C) : L = L' \circ S$

Falls alle Prädikate strukturelle Gleichheit erzwingen, dann ist $\mu SS(C)$ entweder leer oder enthält eine Vereinfachung von C.

Lösbare Restriktionsprobleme, die $p(\overline{\tau}) \in C \Rightarrow \tau_i \overset{\text{ss}}{\sim} \tau_j$ erfüllen, kann man auf einfache Weise in äquivalente atomare Restriktionsprobleme transformieren, indem man die Normalform bzgl. $\longrightarrow\!\!\!\!\!\rightarrow^*$ berechnet. Falls die Normalform nicht-atomare Restriktionen enthält, dann ist C unlösbar.

Korrollar 5.11. *In einem System induktiv definierter, strukturelle Gleichheit erzwingender Prädikate besitzt jeder typisierbare Ausdruck einen allgemeinsten Typ mit atomarer Restriktionsmenge.*

Die Berechnung von $\mu SS(C)$ mit anschließender Transformation in $\longrightarrow\!\!\!\!\!\rightarrow$-Normalform ist natürlich relativ ineffizient, da in der Regel viele Substitutionen in $\mu SS(C)$ zu unlösbaren Restriktionsproblemen führen. Darüber hinaus läßt sich diese Methode

auch nicht direkt auf reguläre Terme übertragen. Glücklicherweise lassen sich jedoch die Berechnung struktureller Ähnlichkeit erzwingender Substitutionen und Ersetzung von Restriktionsmengen gemäß $\longrightarrow\!\!\!\!\!\rightarrow$ in einen einzigen Transformationsprozeß integrieren.

Abbildung 5.3 enthält eine Transformationsrelation $\Longrightarrow_2$, die das Gewünschte leistet. Die Regeln L, Z und S wurden unverändert aus $\Longrightarrow_1$ übernommen, Regel A dagegen wurde in zwei Regeln aufgeteilt: Regel E bewirkt Ersetzungen gemäß der Ersetzungsrelation $\longrightarrow\!\!\!\!\!\rightarrow$, Regel A sorgt für Anpassung der Struktur der Argumente struktureller Ähnlichkeit erzwingender Prädikate.

Grundlage dieser Aufteilung ist die folgende Beobachtung: Angenommen die Restriktionsmenge enthält eine Restriktion der Form $p(\overline{\tau_n})$ und es gibt eine Variable α sowie einen Typausdruck τ der Form $f(\overline{\tau'_m})$, die als Argumente des Prädikats vorkommen. Falls überhaupt eine Lösung L der Restriktionsmenge existiert, dann muß es einen Typausdruck τ' geben, der strukturell ähnlich zu τ ist, sodaß $L(\alpha)$ eine Instanz von τ' ist. Einen solchen Typausdruck könnte man als strukturelles Muster von τ bezeichnen. Man erhält ein solches Muster durch Ersetzen jedes Vorkommens einer Typvariable α oder eines null-stelligen Typkonstruktors $f \in F_0$ durch eine neue, sonst nicht verwendete Typvariable und durch beliebiges Umbenennen von Konstruktoren $f \in F_+$ aus τ. Beispielsweise ist $list(\alpha \times \beta)$ ein strukturelles Muster für $set(int \to \alpha)$. Durch sukzessives Aufzählen und Anwenden aller möglichen Ersetzungen geeigneter Typvariablen durch strukturelle Muster, kann man das Restriktionsproblem in endliche viele neue Restriktionsprobleme transformieren, deren Gesamtheit die Menge aller möglichen Lösungen des ursprünglichen Problems darstellt. Dieses Vorgehen wird solange wiederholt, bis eine Ersetzungsregel aus $\longrightarrow\!\!\!\!\!\rightarrow$ angewendet werden kann. Ein typisches Beispiel für die Vorgehensweise liefert die Konversionsrelation $\lhd$:

$$\begin{aligned} \alpha \lhd (\beta \times \gamma) &\Longrightarrow \alpha_1 \times \alpha_2 \lhd \beta \times \gamma,\ \alpha = \alpha_1 \times \alpha_2 \\ &\Longrightarrow \alpha_1 \lhd \beta,\ \alpha_2 \lhd \gamma,\ \alpha = \alpha_1 \times \alpha_2 \end{aligned}$$

Im Fall struktureller Gleichheit erzwingender Prädikate erübrigt sich das Aufspalten in mehrere neue Probleme, da es, bis auf Variablenumbenennungen, nur ein Muster für τ geben kann, daß strukturell äquivalent zu τ ist. Beispielsweise ist $list(\alpha \times \beta)$ das einzige Muster für $list(\gamma \times int)$.

Lemma 5.12. *Sei $C = C' \cup p(\overline{\tau_n})$ ein Restriktionsproblem, p ein strukturelle Ähnlichkeit erzwingendes Prädikat, $\alpha, \tau \in args(p(\overline{\tau_n}))$ und es gelte $\alpha \in \mathcal{V}$, $root(\tau) \in F_+$. Dann gibt es endliche Menge minimaler Terme $\mathcal{M}_{ss}(\tau, \mathcal{V}(C))$, sodaß für jede Lösung $L \in \mathcal{L}(C)$ ein $\tau' \in \mathcal{M}_{ss}(\tau, \mathcal{V}(C))$ existiert, mit $\tau' \overset{ss}{\sim} \tau$ und $\tau' \leq L(\alpha)$.*

Beweis. Durch strukturelle Induktion über τ zeigt man, daß die Funktion $\mathcal{M}_{ss}$ mit der Definition

$$\mathcal{M}_{ss}(\tau, W) = \begin{cases} \{\beta\} & \text{falls } \tau \in \mathcal{V} \cup F_0 \text{ und } \beta \in \mathcal{V} - W \\ \{g(\tau'_1, \ldots, \tau'_m) \mid \tau'_i \in X_i,\ g \in F_m\} & \\ \qquad \text{falls } \tau = f(\tau_1, \ldots, \tau_m) \text{ und} & \\ \qquad \text{für } i = 1..m \text{ gilt } X_i = \mathcal{M}_{ss}(\tau_i, W_{i-1}) & \\ \qquad \text{und } W_i = W_{i-1} \cup \mathcal{V}(X_i), \text{ wobei } W_0 = W & \end{cases}$$

eine vollständige Menge $M = \mathcal{M}_{ss}(\tau, W)$ von Termen berechnet, sodaß für alle $t \overset{ss}{\sim} L(\tau)$ ein $\tau' \in M$ existiert, mit $\tau' \leq t$ und $\mathcal{V}(W) \cap \mathcal{V}(\tau) = \emptyset$.

Für $\tau \in F_0 \cup \mathcal{V}$ gilt offensichtlich $\beta \leq t$. Wegen $\beta \in \mathcal{V} - W$ folgt die Behauptung unmittelbar. Falls τ von der Form $f(\tau_1, \ldots, \tau_m)$ ist, dann gilt $t = f(t_1, \ldots, t_m)$ für ein $f \in F_m$ aufgrund der Definition von struktureller Ähnlichkeit. Nach Induktionsvoraussetzung existieren $\tau'_1 \in X_1, \ldots, \tau'_m \in X_m$ und Substitutionen $R_1, \ldots, R_m$ mit $t_i = R_i(\tau'_i)$ und $dom(R_i) = \mathcal{V}(\tau'_i)$ für alle i. Nach Definition von $\mathcal{M}_{ss}$ gilt außerdem $\mathcal{V}(\tau'_i) \cap \mathcal{V}(\tau'_j)$ für $i \neq j$ und daher $t = R(g(\tau'_1, \ldots, \tau'_m))$, wobei $R = R_1 \circ \cdots \circ R_m$. Daß die Menge M nur minimale Terme enthält, d.h. $\forall t, t \in M : t \not\leq t'$, ist offensichtlich. □

Für strukturelle Gleichheit erzwingende Prädikate erhält man sogar die schärfere Aussage:

Lemma 5.13. *Sei $C = C' \cup p(\overline{\tau_n})$ ein Restriktionsproblem, p ein strukturelle Gleichheit erzwingendes Prädikat, $\alpha, \tau \in args(p(\overline{\tau_n}))$ und es gelte $\alpha \in \mathcal{V}$, $root(\tau) \in F_+$. Dann gibt es ein $\tau' = \mathcal{M}_{se}(\tau, \mathcal{V}(C))$, sodaß $\tau' \overset{ss}{\sim} \tau$ und $\tau' \leq L(\alpha)$ für jede Lösung $L \in \mathcal{L}(C)$.*

L	$C \cup \{\tau = \tau\}$	$\Longrightarrow_2 C$
Z	$C \cup \{f(\overline{\tau_n}) = f(\overline{\tau'_n})\}$	$\Longrightarrow_2 C \cup \{\tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n\}$
S	$C \cup \{\alpha = \tau\}$	$\Longrightarrow_2 \{\alpha \mapsto \tau\}(C) \cup \{\alpha = \tau\}$ falls $\alpha \in \mathcal{V}(C) - \mathcal{V}(\tau),\ \tau \in \mathcal{V} \Rightarrow \tau \in \mathcal{V}(C)$
E	$C \cup \{p(f_1(\overline{\tau}_1), \ldots, f_n(\overline{\tau}_n))\}$	$\Longrightarrow_2 C \cup S(\overline{r_k})$ falls $p(f_1(\overline{\alpha}_1), \ldots, f_n(\overline{\alpha}_n)) \longrightarrow\!\!\!\!\!\rightarrow \overline{r_k} \in R$ und $S = \{\tau_{11}/\alpha_{11}, \ldots, \tau_{nm}/\alpha_{nm}\}$
A	$C \cup \{p(\overline{\tau_n})\}$	$\Longrightarrow_2 \{\alpha \mapsto \tau'\}(C \cup \{p(\overline{\tau_n})\}) \cup \{\alpha = \tau'\}$ falls $\alpha, \tau \in \mathit{args}(p(\overline{\tau_n})),\ \mathit{root}(\tau) \in F_+$ und $\tau' \in \mathcal{M}_{ss}(\tau, \mathcal{V}(C) \cup \mathcal{V}(\overline{\tau_n}))$

Abbildung 5.3: Transformation struktureller Ähnlichkeit erzwingender Prädikate in atomare Restriktionsmengen

Beweis. Analog zum Beweis von Lemma 5.12. Die Funktion $\mathcal{M}_{se}$ ist dabei wie folgt definiert:

$$\mathcal{M}_{se}(\tau, W) = \begin{cases} \beta & \text{falls } \tau \in \mathcal{V} \cup F_0 \text{ und } \beta \in \mathcal{V} - W \\ f(\tau'_1, \ldots, \tau'_m) & \\ \quad \text{falls } \tau = f(\tau_1, \ldots, \tau_m) \text{ und} & \\ \quad \text{für } i = 1..m \text{ gilt } \tau'_i = \mathcal{M}_{se}(\tau_i, W_{i-1}) & \\ \quad \text{und } W_i = W_{i-1} \cup \mathcal{V}(\tau'_i), \text{ wobei } W_0 = W & \end{cases}$$

□

Satz 5.14. *$\Longrightarrow_2$ ist wohldefiniert und atomar vollständig. Falls C eine Normalform unter $\Longrightarrow_2$ ist, dann ist C entweder unlösbar oder in atomar gelöster Form.*

Beweis. Da die Regeln L, Z und S identisch zu den entsprechenden Regeln der Transformationsrelation $\Longrightarrow_1$ sind, genügt die Betrachtung der Regeln E und A.

Wohldefiniertheit.

E: Zu zeigen ist $L \in \mathcal{L}(C \cup S(\overline{r_k})) \Rightarrow L \in \mathcal{L}(C \cup \{p(f_1(\overline{\tau}_1), \ldots, f_n(\overline{\tau}_n))\})$. Nach Definition $\longrightarrow\!\!\!\!\!\rightarrow$ gilt $L(S(p(f_1(\overline{\alpha}_1), \ldots, f_n(\overline{\alpha}_n)))) \longrightarrow\!\!\!\!\!\rightarrow L(S(\overline{r_k}))$. Aus $L(S(\overline{r_k})) \longrightarrow\!\!\!\!\!\rightarrow^* \emptyset$ folgt $L(p(f_1(\overline{\tau}_1), \ldots, f_n(\overline{\tau}_n))) \longrightarrow\!\!\!\!\!\rightarrow^* \emptyset$ wegen $S(\alpha_{ij}) = \tau_{ij}$.

A: Zu zeigen ist $L \in \mathcal{L}(\{\alpha \mapsto \tau'\}(C \cup \{p(\overline{\tau_n})\}) \cup \{\alpha = \tau'\}) \Rightarrow L \in \mathcal{L}(C \cup \{p(\overline{\tau_n})\})$. Da $L(\alpha) = L(\tau')$ gilt $L(C) = L(\{\alpha \mapsto \tau'\}C)$ und $L(p(\overline{\tau_n})) = L(\{\alpha \mapsto \tau'\}p(\overline{\tau_n}))$ woraus sofort $L(C \cup \{p(\overline{\tau_n})\}) \longrightarrow\!\!\!\!\!\rightarrow^* \emptyset$ folgt.

Vollständigkeit. Sei $C = C_0 \cup \{p(\overline{\tau_n})\}$ und $L \in \mathcal{L}(C)$. Zu zeigen ist, daß eine Transformationsregel $C \overset{x}{\Rightarrow} C'$ existiert, sodaß $L = L'_{|\mathcal{V}(C)}$ für ein $L' \in \mathcal{L}(C')$. Falls $p(\overline{\tau_n})$ von der Form $p(f_1(\overline{\tau}_1), \ldots, f_n(\overline{\tau}_n))$ für $f_1, \ldots, f_n \in F_+$, dann gibt es ein Paar $p(f_1(\overline{\alpha}_1), \ldots, f_n(\overline{\alpha}_n)) \longrightarrow\!\!\!\!\!\rightarrow \overline{r_k} \in R$, sodaß $L(p(f_1(\overline{\tau}_1), \ldots, f_n(\overline{\tau}_n))) \longrightarrow\!\!\!\!\!\rightarrow L(S(\overline{r_k})) \longrightarrow\!\!\!\!\!\rightarrow^* \emptyset$. Aus $C \cup \{p(\overline{\tau_n})\} \overset{E}{\Rightarrow} C \cup S(\overline{r_k})$ folgt $L' = L$. Falls $p(\overline{\tau_n})$ nicht von der Form $p(f_1(\overline{\tau}_1), \ldots, f_n(\overline{\tau}_n))$ ist, muß es $\alpha, \tau \in \tau_1, \ldots, \tau_n$ geben, sodaß $\alpha \in \mathcal{V}$ und $root(\tau) \in F_+$. Da p nach Voraussetzung strukturelle Ähnlichkeit erzwingt, gilt $L(\alpha) \overset{ss}{\sim} L(\tau)$. Aufgrund von Lemma 5.12 gibt es ein strukturelles Muster $\tau' \overset{ss}{\sim} \tau$, sodaß $\tau' \leq L(\alpha)$. Also gibt es eine Substitution S mit $dom(S) = \mathcal{V}(\tau')$, sodaß $L(\alpha) = S(\tau')$. Wählt man nun $L' = L \circ S$, dann gilt $L = L'_{|\mathcal{V}(C)}$, $L'(\alpha) = L'(\tau')$, $L'(p(\overline{\tau_n})) = L(p(\overline{\tau_n}))$, sowie $L \circ \{\alpha \mapsto \tau'\} = L'$ und daher $L' \in \mathcal{L}(\{\alpha \mapsto \tau'\}(C \cup \{p(\overline{\tau_n})\}) \cup \{\alpha = \tau'\})$.

Sei C eine Normalform unter $\Longrightarrow_2$. C läßt sich als Vereinigung von Gleichungen $G = \{s_1 = t_1, \ldots, s_n = t_n\}$ und Restriktionen $R = \{p_1(\overline{u}_1), \ldots, p_l(\overline{u}_l)\}$ darstellen. Aufgrund von Satz 5.5 erfüllt G die Bedingungen 1, 2 und 3 der Definition 5.2. Darüber hinaus gilt sicher $s_i \notin \mathcal{V}(R)$, da sonst Regel S angewendet werden könnte. Sei $p(\overline{\tau_n})$ eine Restriktion aus R. Regel A impliziert, daß entweder $\tau_i \in F_0 \cup \mathcal{V}$ für alle τ_i, oder $root(\tau_i) \in F_+$ für alle τ_i gilt. Im ersten Fall erfüllt C Bedingung 4 der Definition 5.2, im zweiten Fall ist $p(\overline{\tau_n})$ von der Form $p(f_1(\overline{\tau}_1), \ldots, f_n(\overline{\tau}_n))$ und es gibt keine Regel $p(f_1(\overline{\alpha}_1), \ldots, f_n(\overline{\alpha}_n)) \longrightarrow\!\!\!\!\!\rightarrow \overline{r_k}$. In diesem Fall hat C keine Lösung. Falls $\tau_i \in F_0$ für alle i, dann gibt es keine Lösung für C, denn sonst wäre Regel E anwendbar. Somit folgt insgesamt, daß jede Normalform unter $\Longrightarrow_2$ entweder unlösbar oder in atomar gelöster Form ist. □

Leider ist $\Longrightarrow_2$ keine terminierende Transformationsrelation, wie das folgende Beispiel zeigt. Das Restriktionsproblem $\alpha \lhd l(\beta),\ \beta \lhd \alpha$ führt nämlich unausweichlich zu

einer unendlichen Transformationssequenz.

$$\begin{aligned}
\alpha \lhd l(\beta),\ \beta \lhd \alpha &\overset{\text{A}}{\Longrightarrow} l(\alpha_1) \lhd l(\beta),\ \beta \lhd l(\alpha_1),\ \alpha = l(\alpha_1) \\
&\overset{\text{E}}{\Longrightarrow} \alpha_1 \lhd \beta,\ \beta \lhd l(\alpha_1),\ \alpha = l(\alpha_1) \\
&\overset{\text{A}}{\Longrightarrow} \alpha_1 \lhd l(\beta_1),\ l(\beta_1) \lhd l(\alpha_1),\ \alpha = l(\alpha_1),\ \beta = l(\beta_1) \\
&\overset{\text{E}}{\Longrightarrow} \alpha_1 \lhd l(\beta_1),\ \beta_1 \lhd \alpha_1,\ \alpha = l(\alpha_1),\ \beta = l(\beta_1) \\
&\ \ \vdots
\end{aligned}$$

Das obige Problem ist unlösbar, da jede Lösung L die Äquivalenz $L(\alpha) \overset{\text{ss}}{\sim} L(\beta)$ respektieren muß, ebenso wie $L(\alpha) \overset{\text{ss}}{\sim} L(l(\beta))$. Aus der Transitivität von $\overset{\text{ss}}{\sim}$ folgt aber auch $L(\beta) \overset{\text{ss}}{\sim} L(l(\beta))$, was aber für endliche Terme unmöglich ist.

Um unlösbare Restriktionsprobleme zu erkennen, muß man ein Äquivalent zu dem sogenannten „occur-check“, $\alpha \notin \mathcal{V}(\tau)$ in Regel S, zu den Transformationsregeln hinzufügen. Wie man am obigen Beispiel leicht erkennt, reicht eine naive Ausdehnung des „occur-checks“, etwa in der Form $\alpha \notin \mathcal{V}(\tau)$ als zusätzliche Vorbedingung der Transformationsregel A, nicht aus.[5]

Jedes Restriktionsproblem C über strukturelle Ähnlichkeit erzwingenden Prädikaten induziert eine Äquivalenzrelation $\overset{C}{\sim}$ auf $\mathcal{V}(C)$.

Definition 5.15. Sei C ein Restriktionsproblem. $\overset{C}{\sim}$ ist der reflexive, transitive und symmetrische Abschluß der Relation

$$\begin{aligned}
EQ(C) = &\bigcup\{sim(\tau, \tau') \mid \tau = \tau' \in C\} \\
&\cup \bigcup\{sim(\tau_i, \tau_{i+1}) \mid p(\overline{\tau_n}) \in C,\ i \in 1..n-1\}
\end{aligned}$$

wobei

$$sim(\tau, \tau') = \begin{cases} \{(\tau, \tau')\} & \text{falls } \tau, \tau' \in \mathcal{V} \\ \bigcup_{i=1}^{n} sim(\rho_i, \rho_i') & \text{falls } \tau = f(\overline{\rho_n}), \tau' = g(\overline{\rho_n'}) \\ \emptyset & \text{sonst} \end{cases}$$

Hilfssatz 5.16. *Seien $\alpha, \beta \in \mathcal{V}(C)$ und $\alpha \overset{C}{\sim} \beta$, dann gilt: $S \in \mathcal{L}(C) \Rightarrow S(\alpha) \overset{ss}{\sim} S(\beta)$.*

[5] Dies wurde auch schon in früheren Arbeiten zum Thema Konversiontypisierung erkannt. Unsere Darstellung weicht jedoch in einigen Details, wegen der Verallgemeinerung auf Prädikate beliebiger Stelligkeit, von den Darstellungen in [Mit84] bzw. [FM88] ab.

Beweis. Folgt sofort aus der Definition von $\overset{ss}{\sim}$ und $\overset{C}{\sim}$. □

Lemma 5.17. *Sei $\equiv$ eine Äquivalenzrelation mit der Eigenschaft*

$$\forall \alpha, \beta \in \mathcal{V}(C) : \alpha \equiv \beta \Rightarrow \forall L \in \mathcal{L}(C) : L(\alpha) \overset{ss}{\sim} L(\beta)) \ . \tag{5.4}$$

Für $\alpha \in \mathcal{V}$ bezeichne $[\alpha]_{\equiv}$ die Äquivalenzklasse von α in $\equiv$, für $\tau \notin \mathcal{V}$ sei $[\tau]^E$ die Vereinigung aller Äquivalenzklassen der in τ vorkommenden Variablen.

$$\begin{aligned} [\alpha]_{\equiv} &= \{\, \beta \mid \alpha \equiv \beta \,\} \\ [\tau]^{\equiv} &= \{\, [\alpha]_{\equiv} \mid \alpha \in \mathcal{V}(\tau) \,\} \end{aligned}$$

Falls C eine Restriktion $p(\overline{\tau_n})$ enthält, oder eine Gleichung $\tau_1 = \tau_2$, sodaß $[\alpha]_{\equiv} \in [\tau]^{\equiv}$ für $\alpha, \tau \in \{\tau_1, \ldots, \tau_n\}$, dann hat C keine Lösung.

Beweis. Wir nehmen an es gäbe eine solche Substitution. Da p nach Definition strukturelle Ähnlichkeit erzwingt, muß für jede erfüllende Substitution S auch $S(\alpha) \overset{ss}{\sim} S(\tau)$ gelten. Andererseits folgt aus $\alpha \overset{E}{\sim} \beta$ aber auch $S(\alpha) \overset{ss}{\sim} S(\beta)$ und wegen der Transitivität von $\overset{ss}{\sim}$ gilt auch $S(\beta) \overset{ss}{\sim} S(\tau)$. Da β ein Subterm von τ ist, müßte $S(\tau)$ einen zu sich selbst strukturell ähnlichen Subterm enthalten, was aber nach Definition der strukturellen Ähnlichkeit unmöglich ist. □

Man beachte, daß die Bedingung $[\alpha]_{\equiv} \in [\tau]^{\equiv}$ den normalen „occur-check“ $\alpha \in \mathcal{V}(\tau)$ umfaßt. Darüber hinaus gilt Lemma 5.17 auch für strukturelle Gleichheit erzwingende Prädikate (siehe Korollar 5.7).

Diese Beobachtung kann man benutzen, um eine terminierende Transformationsrelation $\Longrightarrow_3$ zu konstruieren, die in Abbildung 5.4 angegeben ist. Der wesentliche Unterschied zur Relation $\Longrightarrow_2$ besteht darin, daß immer ganze Klassen strukturell äquivalenter Variablen durch entsprechende strukturelle Muster ersetzt werden, wie sie durch die Äquivalenzrelation E indiziert werden. Zur Vereinfachung wurde die Regel S aus $\Longrightarrow_2$ in zwei Regeln aufgeteilt: S_1 ersetzt die Variable α durch β falls beide Variablen in der Restriktionsmenge C vorkommen. Anwendung von S_2 führt zu einer Ersetzung aller Variablen der Äquivalenzklasse $[\alpha]_E$ durch ein Muster, das strukturell äquivalent zu τ ist. Nach dieser Ersetzung sind alle Variablen aus $[\alpha]_E$ in der resultierenden Restriktionsmenge gelöst. Dabei sei $\mathcal{SS}(\tau, \{\alpha_1, \ldots, \alpha_n\}, W)$ eine vollständige Menge minimaler Substitutionen die keine Variablen aus W involvieren, sodaß für

jedes α_i, $S(\alpha_i)$ ein strukturelles Muster für τ ist und außerdem $\mathcal{V}(S(\alpha_i)) \neq \mathcal{V}(S(\alpha_j))$ für alle $i \neq j \in 1..n$.

$\Longrightarrow_3$ verwendet darüberhinaus zur frühzeitigen Erkennung unlösbarer Restriktionsprobleme ein spezielles Restriktionsproblem $\top$ als Repräsentant der Klasse aller unlösbaren Restriktionsprobleme.

Zur Manipulation von Äquivalenzrelationen verwenden wir die folgende Notation:

E_α	ist die Äquivalenzrelation E ohne die Äquivalenzklasse $[\alpha]_E$.
$\{\beta/\alpha\}E$	ist die Äquivalenzrelation E die man durch Umbenennung von α in β erhält. Diese Operation ist nur definiert, falls $\alpha \in [\beta]_E$.
$E + R$	ist der transitive und symmetrische Abschluß der Relation $E \cup R$.
$\|E\|$	ist die Anzahl der Äquivalenzklassen in E.

Satz 5.18. *$\Longrightarrow_3$ ist wohldefiniert, vollständig und terminiert falls die Äquivalenzrelation E die Gleichung 5.4 erfüllt. Eine Normalform unter $\Longrightarrow_3$ ist entweder $\top$ oder in atomar gelöster Form.*

Beweis. Man beachte, daß falls E die Gleichung 5.4 erfüllt und $E, C \Longrightarrow_3 E', C'$ gilt, E' auch die Gleichung 5.4 erfüllt. Die Wohldefiniertheit von $\Longrightarrow_3$ folgt aus der Wohldefiniertheit von $\Longrightarrow_2$, da jede Transformationssequenz in $\Longrightarrow_3$, die nicht die Regeln F_1, F_2, Z_1 oder Z_2 involviert, auf eine Transformationssequenz in $\Longrightarrow_2$ abgebildet werden kann, wenn man die Äquivalenzrelationskomponente ignoriert und Verwendungen der Regeln S_2 und A in Sequenzen von S- und A-Regeln aus $\Longrightarrow_2$ übersetzt. Die Vollständigkeit basiert auf Lemma 5.17. Termination folgt durch noethersche Induktion über das Tripel $\langle |E|, U(C), S(C)\rangle$, wobei E, $U(C)$ und $S(C)$ wie für $\Longrightarrow_1$ definiert sind. □

Die Transformationsrelationen $\Longrightarrow_1$ und $\Longrightarrow_3$ können kombiniert werden, wenn sich die Prädikate in zwei disjunkte Mengen unterteilen lassen: Eine Menge P_N nicht rekursiv definierter Prädikate und eine Menge P_{SI} struktureller Ähnlichkeit erzwingender, induktiv definierter Prädikate. Dabei dürfen Prädikate aus P_{SI} auf der rechten

L	$E, C \cup \{\tau = \tau\}$	$\Longrightarrow_3 E, C$
Z	$E, C \cup \{f(\overline{\tau_n}) = f(\overline{\tau'_n})\}$	$\Longrightarrow_3 E, C \cup \{\tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n\}$
T	$E, C \cup \{p(f_1(\overline{\tau}_1), \ldots, f_n(\overline{\tau}_n))\}$	$\Longrightarrow_3 E, C \cup S(\overline{r_k})$
		falls $p(f_1(\overline{\alpha}_1), \ldots, f_n(\overline{\alpha}_n)) \longrightarrow\!\!\!\!\!\rightarrow \overline{r_k} \in R$
		und $S = \{\alpha_{11} \mapsto \tau_{11}, \ldots, \alpha_{nm} \mapsto \tau_{nm}\}$
S_1	$E, C \cup \{\alpha = \beta\}$	$\Longrightarrow_3 \{\beta/\alpha\}(E + \{(\alpha, \beta)\}), \{\alpha \mapsto \beta\}(C) \cup \{\alpha{=}\beta\}$
		falls $\alpha, \beta \in \mathcal{V}(C),\ \alpha \neq \beta$
S_2	$E, C \cup \{\alpha = \tau\}$	$\Longrightarrow_3 E', S \cup (\vec{S} \circ \{\alpha \mapsto \tau\})(C) \cup \{\alpha = \tau\}$
		falls $\alpha \in \mathcal{V}(C),\ \tau \notin \mathcal{V},\ [\alpha]_E \notin [\tau]^E$
		und $\vec{S} \in \mathcal{SS}(\tau, [\alpha]_E - \{\alpha\}, \mathcal{V}(C \cup \{\alpha = \tau\}))$
		wobei $E' = E_\alpha + EQ(\{\, \tau = \vec{S}(\beta) \mid \beta \in [\alpha]_E \,\})$
A	$E, C \cup \{p(\overline{\tau_n})\}$	$\Longrightarrow_3 E', S \cup \vec{S}(C \cup \{p(\overline{\tau_n})\})$
		falls $\alpha, \tau \in \mathit{args}(p(\overline{\tau_n}))$
		sowie $\mathit{root}(\tau) \in F_+,\ [\alpha]_E \notin [\tau]^E$
		und $\vec{S} \in \mathcal{SS}(\tau, [\alpha]_E, \mathcal{V}(C \cup \{p(\overline{\tau_n})\}))$
		wobei $E' = E_\alpha + EQ(\{\, \tau = \vec{S}(\beta) \mid \beta \in [\alpha]_E \,\})$
F_1	$E, C \cup \{f(\overline{\tau_n}) = g(\overline{\tau'_n})\}$	$\Longrightarrow_3 E, \top$
F_2	$E, C \cup \{p(f_1(\overline{\tau}_1), \ldots, f_n(\overline{\tau}_n))\}$	$\Longrightarrow_3 E, \top$
		falls $\nexists\, p(f_1(\overline{\alpha}_1), \ldots, f_n(\overline{\alpha}_n)) \longrightarrow\!\!\!\!\!\rightarrow \overline{r_m} \in R$
Z_1	$E, C \cup \{\alpha = \tau\}$	$\Longrightarrow_3 E, \top$
		falls $\tau \notin \mathcal{V},\ [\alpha]_E \in [\tau]^E$
Z_2	$E, C \cup \{p(\overline{\tau_n})\}$	$\Longrightarrow_3 E, \top$
		falls $\exists \alpha, \tau \in \mathit{args}(p(\overline{\tau_n})),\ \tau \notin \mathcal{V},\ [\alpha]_E \in [\tau]^E$

Abbildung 5.4: Terminierende Transformation in atomare Restriktionsmengen

Seite von Ersetzungsregeln für Prädikate aus P_N verwendet werden, aber nicht umgekehrt. Typische Beispiele für solche Prädikate sind $p_\in$ und $p_\downarrow$ aus Abbildung 5.1.

Sei $\Longrightarrow_{ns}$ die Menge von Ersetzungsregeln, die man erhält, wenn man die Verwendung von $\Longrightarrow_3$-Regeln auf Prädikate aus P_{SI} einschränkt und die Regel

$$\begin{aligned}\mathrm{A_n}\ E, C \cup \{p(\overline{\tau_n})\} \Longrightarrow_{ns}\ & E, C \cup \overline{r_m} \cup \{\tau_1 = l_1, \ldots, \tau_n = l_n\} \\ & \text{falls}\ \ p \in P_N,\ p(\overline{l_n}) \longrightarrow\!\!\!\!\!\rightarrow \overline{r_m} \\ & \text{eine Ersetzungsregel ist,} \\ & \text{entfernt von } \mathcal{V}(C) \cup \mathcal{V}(\overline{\tau_n})\end{aligned}$$

hinzufügt.

Satz 5.19. *$\Longrightarrow_{ns}$ ist wohldefiniert, vollständig und terminierend. Eine Normalform unter $\Longrightarrow_{ns}$ ist entweder $\top$ oder in atomar gelöster Form.*

Beweis. Wohldefiniertheit und Vollständigkeit sind klar. Termination folgt durch noethersche Induktion über das Komplexitätsmaß $\langle B(C), |E|, U(C), S(C)\rangle$. Der zweite Teil des Satzes gilt offensichtlich. □

Damit haben wir gezeigt, daß ein Restriktionsproblem C über nicht rekursiven Prädikaten, und strukturelle Ähnlichkeit erzwingenden Prädikaten, in eine Menge atomar gelöster Restriktionsmengen $C_1, \ldots, C_n$ transformiert werden kann, deren Lösbarkeit die Lösbarkeit der ursprünglichen Restriktionsmenge impliziert. Das Supremum der genauesten Vereinfachungen der lösbaren Restriktionsmengen C_i ist dann eine genaueste Vereinfachung der ursprünglichen Restriktionsmenge C.

5.4 Zerlegbare Prädikate

Es verbleibt noch zu zeigen, daß die Lösbarkeit atomarer Restriktionprobleme entscheidbar ist. O.B.d.A. kann man davon ausgehen, daß C keine erfüllbaren Restriktionen der Form $p(\overline{a_n})$ mit $a_i \in F_0$ enthält. Somit enthält jede Restriktion mindestens eine Typvariable. Man beachte nun, daß die Äquivalenzrelation $\overset{C}{\sim}$ eine Partition der Restriktionsmenge C induziert. Wir schreiben

$$C = \biguplus_{i=1..n}^{\overset{SS}{\sim}} C_i$$

falls $C = C_1 \uplus \cdots \uplus C_n$, wobei $n = |\{[\alpha]_{\lesssim_C} \mid \alpha \in \mathcal{V}(C)\}|$ und $\mathcal{V}(C_i) = [\alpha]_{\lesssim_C}$ für eine $\alpha \in \mathcal{V}(C)$. Jedes der C_i kann unabhängig gelöst werden, und die Lösungen können zu einer Lösung für C verknüpft werden. Somit verbleiben zwei Sorten von Restriktionsmengen: solche die Basistypen und Typvariablen enthalten und solche die nur Variablen enthalten. Restriktionsprobleme, die Basistypen enthalten, besitzen nur Basistyplösungen, d.h. $L \models C \Rightarrow L \in \mathcal{V} \to F_0$. Restriktionsprobleme, die nur Typvariablen enthalten, führen zu einer Komplikation bei der Lösungssuche: sie können u.U. eine Lösung besitzen, ohne daß eine Basistyplösung existiert.

Als ein Beispiel für dieses Phänomen betrachten wir strukturelle Konversionen, angereichert mit Prädikaten für zwei überladene Operatoren p und q, mit den Ersetzungregeln

$$\begin{aligned} p(a \times b) &\longrightarrow\!\!\!\!\!\rightarrow q(a), q(b) \\ q(int) &\longrightarrow\!\!\!\!\!\rightarrow \emptyset \end{aligned}$$

Das Restriktionsproblem $\{\alpha \lhd \beta,\ \beta \lhd \gamma, p(\gamma)\}$ besitzt zwar die Lösung

$$\{\alpha \mapsto int \times int,\ \beta \mapsto int \times int,\ \gamma \mapsto int \times int\}\ ,$$

aber keine Basistyplösung!

Obwohl man in diesem Beispiel sofort eine Lösung angeben kann, ist a priori nicht klar, daß man die Lösungssuche nach endlich vielen Schritten abbrechen kann, wenn keine Lösung gefunden wurde. Wir betrachten daher zunächst eine spezielle Darstellung für Lösungen.

Lösungen für atomare Restriktionsprobleme über induktiv definierten, strukturelle Ähnlichkeit erzwingenden Prädikaten kann man als Komposition *partieller* Lösungen darstellen.

Definition 5.20. Eine Substitution S heißt *partielle Lösung* für C_0 mit äußeren Konstruktoren $f_1, \ldots, f_k \in F_n$, falls

1. $dom(S) = \mathcal{V}(C_0) = \{\alpha_1, \ldots, \alpha_k\}$
2. $\forall \alpha_i : \exists \beta_1^i, \ldots, \beta_n^i : S(\alpha_i) = f_i(\beta_1^i, \ldots, \beta_n^i)$,
3. $\forall p(\alpha_{i_1}, \ldots, \alpha_{i_k}) \in C_0 : p(f_{i_1}(\overline{\gamma}_1), \ldots, f_{i_k}(\overline{\gamma}_k)) \longrightarrow\!\!\!\!\!\rightarrow \overline{r_m} \in R$

Falls S eine partielle Lösung C_0 ist, kann $S(C_0)$ ersetzt werden durch $\biguplus^{\text{ss}}_{i=1..l} C_i$. Falls außerdem noch Lösungen $L_1, \ldots, L_l$ für $C_1, \ldots, C_l$ existieren, dann ist $S \circ L_1 \circ \cdots \circ L_l$ eine Lösung für C_0.

Partielle Lösungen können auf einfache Weise als Bäume gezeichnet werden. Die Knoten dieser Bäume sind Restriktionsmengen, die Äste markieren Substitutionen bzw. Ersetzungen. Das allgemeine Schema ist in Abbildung 5.5 angegeben, der Lösungsbaum in Abbildung 5.6 beschreibt eine Lösung für die Restriktionsmenge $\{\alpha \lhd \beta,\ \beta \lhd \gamma, p(\gamma)\}$.

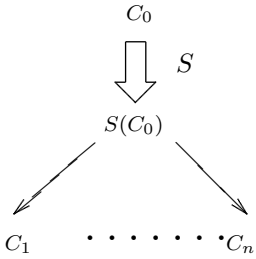


Abbildung 5.5: Lösungsbaumschema

Betrachtet man sich diesen Lösungsbaum genau, so erkennt man, daß alle Restriktionsmengen mit Ausnahme der Blattknoten die gleiche Anzahl von Variablen enthalten. Diese Eigenschaft der Lösungsbäume kann durch eine besondere Form der Ersetzungsregeln garantiert werden.

Definition 5.21. Ein System induktiv definierter Prädikate heißt zerlegbar, falls die Erfüllbarkeit durch links-lineare Ersetzungsregeln der Form

$$p(f_1(\overline{\alpha}_1), \ldots, f_n(\overline{\alpha}_n)) \longrightarrow\!\!\!\!\!\rightarrow \overline{r_k}$$

beschrieben werden kann und die Menge der Äquivalenzklassen der Äquivalenzrelation $\overset{\overline{r_k}}{\sim}$ durch $\{\, \{\alpha_{1i}, \ldots, \alpha_{ni}\} \mid i = 1..m \,\}$ gegeben ist, wobei $f_1, \ldots, f_n \in F_m$.

Man beachte, daß Zerlegbarkeit, im Gegensatz zu struktureller Ähnlichkeit, ein rein syntaktisches Kriterium ist, und daher im Falle von benutzerdefinierbaren Überla-

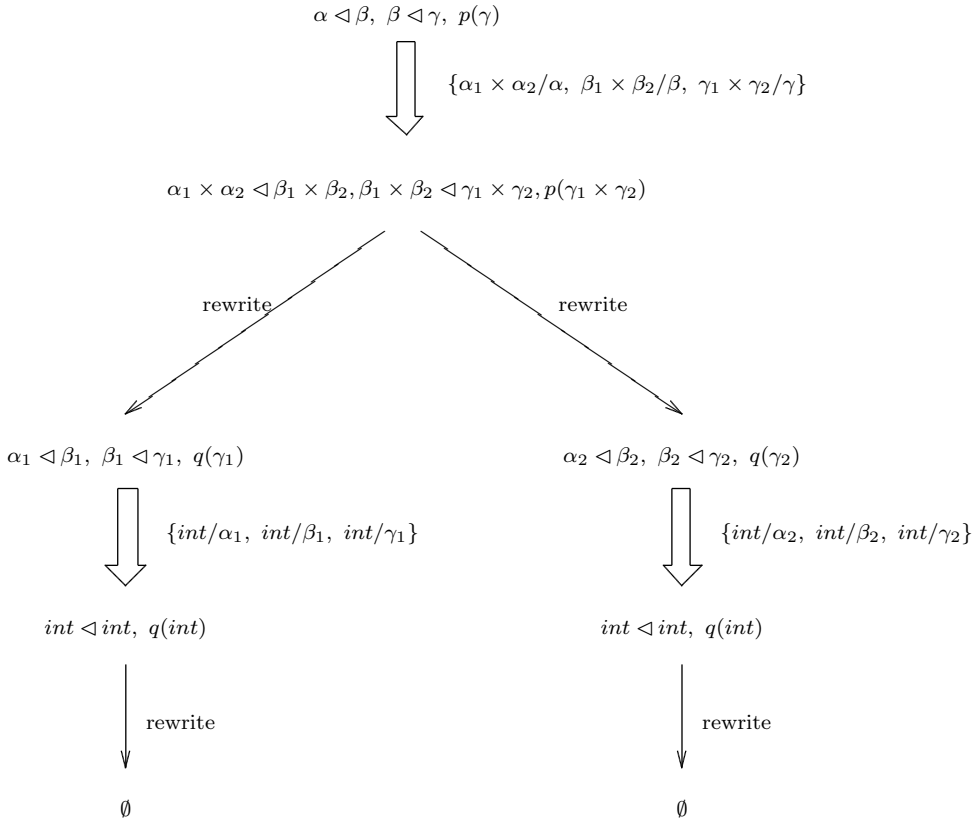


Abbildung 5.6: Ein Lösungsbaum für $\{\alpha \lhd \beta,\ \beta \lhd \gamma, p(\gamma)\}$

dungen bzw. Konversionen von der Typinferenzkomponente leicht überprüft werden kann.

Satz 5.22. *Sei C eine atomare Restriktionsmenge deren Variablen eine einzelne $\overset{C}{\sim}$-Äquivalenzklasse bilden. Es ist entscheidbar, ob C gelöst werden kann.*

Beweis. Zerlegbarkeit impliziert, daß die Menge aller Restriktionsprobleme, die in einem Lösungsbaum auftreten können, endlich ist (modulo Variablenumbenennungen). Jede Applikation einer partiellen Lösung mit äußeren Konstruktoren $f_i \in F_m$ auf eine Restriktionsmenge mit n Variablen, die eine einzelne $\overset{C}{\sim}$-Äquivalenzklasse bilden, führt zu m neuen Restriktionsmengen, die jeweils n neue Variablen enthalten. Daher kann jeder Lösungsbaum so beschnitten werden, daß seine Höhe kleiner als die Anzahl $h(n)$ von Restriktionsmengen über n Variablen ist. □

Außerdem ist entscheidbar, ob C unendlich viele Lösungen hat, nur Basistyplösungen, oder ob eine gegebene Substitution eine genaueste Vereinfachung ist.

Satz 5.23. *Für lösbare Restriktionsmengen über zerlegbaren Prädikaten kann eine genaueste Vereinfachung effektiv berechnet werden.*

Beweis. Aufgrund von Lemma 4.36, Lemma 4.33 und der vorangegangenen Diskussion, genügt es zu zeigen, daß genaueste Vereinfachungen für atomare Restriktionsmengen C, deren Variablen eine einzelne $\overset{C}{\sim}$-Äquivalenzklasse bilden, berechnet werden können. Sei also C eine solche Menge. Falls C einen Basistyp enthält, dann existieren nur endlich viele Lösungen, deren Supremum eine genaueste Vereinfachung für C ist. Falls C nur aus Variablenrestriktionen besteht, müssen die Bedingungen (a) und (b) des Lemmas 4.33 überprüft werden.

Falls Bedingung (a) verletzt wird, d.h. es existieren Variablen α und β, sodaß $L \models C \Rightarrow L(\alpha) = L(\beta)$, dann ist $\{\beta \mapsto \alpha\}$ eine Vereinfachung von C. Diese Vereinfachung führt zu einer neuen Restriktionsmenge mit $n-1$ Variablen, und kann daher höchstens n mal angewendet werden. Aber wie wird (b) überprüft?

Man betrachte einen Lösungsbaum für $C = C_1^0$, sodaß $L(\alpha) \neq L(\beta)$. O.B.d.A. nehme man an, daß die Mengen der Variablen unterschiedlicher Restriktionsmengen des Lösungsbaums keine gemeinsamen Elemente besitzen. Dann muß es einen kürzesten

Pfad $w = j_1, \ldots, j_k$ geben, für den $root(L(\alpha)/w) \neq root(L(\beta)/w)$ gilt. Nun existieren Substitutionen $S_1, \ldots, S_k$, Restriktionsmengen C^i_j für $i \in 1..k$, $j \in 1..n_i$und Lösungen $R^i_j \models C^i_j$ für $i \in 1..k$, $j \in 1..n_i$, $j \neq j_i \vee i = k$ sodaß $L = S_1 \circ \cdots \circ S_k \circ \bigcup_{i,j \neq j_i \vee i=k} R^i_j$ und $S_i(C^{i-1}_{j_{i-1}}) \longrightarrow^* C^i_1 \uplus \cdots \uplus C^i_{n_i}$ (siehe Abbildung 5.7). Die Lösungsbäume für C^i_j, $j \neq j_i \vee i = k$ können in ihrer Höhe auf $h(n)$ beschränkt werden, ebenso wie die Länge des Pfades von C^1_1 nach $C^j_{j_k}$. Also kann die Gesamthöhe des Baumes auf $2h(n)$ eingeschränkt werden. Somit erhält man einen endlichen Suchraum für Bäume, die Bedingung (b) verletzen.

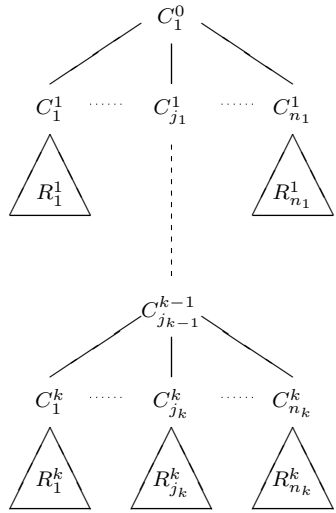


Abbildung 5.7: Ein Lösungsbaum, der 4.33 (b) verletzt.

Bedingung (b) wird folgendermaßen überprüft: Für jede Variable $\alpha \in \mathcal{V}(C)$ und Typkonstruktor $f \in F_m$ wendet man eine Substitution an, die α durch $f(\overline{\beta_m})$ ersetzt, wobei $\overline{\beta_m}$ neue Typvariablen sind. Dann überprüft man die Lösbarkeit der resultierenden Restriktionsmengen. Auf diese Weise erhält man eine Substitution $S = \{\alpha_1 \mapsto f_1(\overline{\beta_m}_1), \ldots, \alpha_n \mapsto f_n(\overline{\beta_m}_n)\}$, sodaß $L \models C \Rightarrow root(L(\alpha_k)) = f_k$. Offensichtlich ist S eine Vereinfachung von C. Falls S die Identität ist, kann C nicht weiter vereinfacht werden. Falls die f_i Basistypen sind, dann hat C nur Basistyplösungen

und somit ist S schon eine genaueste Vereinfachung von C.

Ansonsten ist $m \geq 1$ und man wendet S auf C an, berechnet die Normalformen von $S(C)$ unter der Relation $\Longrightarrow_3$ und wendet die Suche nach der Verletzung von (b) rekursiv auf die lösbare Teilmenge $C_1, \ldots, C_l$ der resultierenden Restriktionsmengen an. Falls dieser Prozeß terminiert, liefert er genaueste Vereinfachungen $A_1, \ldots, A_l$ von $C_1, \ldots, C_l$. Auf der Basis von Lemma 4.36 erhält man $\bigwedge_{i=1..l} A_i \circ S$ als genaueste Vereinfachung von C.

Der gesamte Berechnungsprozeß muß terminieren, da die Lösbarkeit von C eine obere Schranke für die Höhe von $S(\alpha)$ jeder Vereinfachung S von C festlegt: Sei $H(C)$ die kleinste natürliche Zahl m_0, für die eine Lösung $L \models C$ existiert, die $height(L(\alpha)) \leq m_0$ für alle $\alpha \in \mathcal{V}(C)$. Falls $H(C) = 1$ ist, besitzt C eine Basistyplösung. Dies impliziert, daß die obige Substitution S entweder die Identität ist oder C nur Basistyplösungen besitzt. Für $H(C) > 1$ erfüllen alle lösbaren Normalformen C' von $S(C)$ unter $\Longrightarrow_3$ die Bedingung $H(C') \leq H(C) - 1$. □

5.5 Diskussion

Die Komplexität der Berechnung sowohl von Lösungen als auch genauester Vereinfachungen atomarer Restriktionsmengen scheint zu hoch für einen praktikablen Einsatz. Selbst wenn man garantieren kann, daß jedes lösbare Restriktionsproblem Basistyplösungen besitzt, müssen im schlimmsten Fall für ein Restriktionsproblem über n Variablen immer noch $|F_0|^n$ Substitutionen untersucht werden, um die Frage der Lösbarkeit zu entscheiden und eine allgemeinste Vereinfachung zu berechnen. Wir zeigen im nächsten Kapitel, daß man mit Einschränkungen an die Definition der Basistypkonversionsordnung und die Definition überladener Operatoren einen polynomial zeitbeschränkten Algorithmus angeben kann.

Eine weitere Quelle der Ineffizienz des Transformationssystems liegt in der notwendigen Verzweigung bei der Auflösung endlicher Überladungen. Hier wäre ein gangbarer Kompromiss, diese Prädikate erst in einer zweiten Phase zu untersuchen, nachdem man die strukturelle Äquivalenz erzwingenden Prädikate in atomare Teilprobleme transformiert hat und dem Anwender eine nicht auflösbare Überladung als Typfehler zu melden, oder eine der Überladungsinstanzen als Voreinstellung auszuwählen.

Mit diesem Vorgehen ließe sich z.B. der überladene Applikationsoperator von META-IV sehr gut behandeln, der die Verwendung endlicher Abbildungen an Stelle normaler Funktionen in Funktionsapplikationen erlaubt: im Falle eines überladenen Applikationsoperators möchte man sicher nicht bei jeder Applikation zur Laufzeit prüfen, ob die angewendete Funktion eine endliche Abbildung ist. Daher wäre die fehlende Abstrahierbarkeit des Applikationsoperators sicher zu verschmerzen.

Die vorgestellte Erweiterung von strukturellen Äquivalenz erzwingenden Prädikaten hin zu strukturelle Ähnlichkeit erzwingenden Prädikaten scheint in erster Linie von theoretischem Interesse. Einerseits liefert die allgemeinste Vereinfachung eines Restriktionsproblems in der Regel kein atomar gelöstes Restriktionproblem, was die Verständlichkeit solcher Restriktionsprobleme reduziert. Andererseits enthält die Menge der strukturell ähnlichen Muster eines Terms $O(2^{|\tau|\cdot|F|})$ viele Elemente, was zu einer exponentiell anwachsenden Menge zu untersuchender Restriktionsprobleme führt. Im Gegensatz dazu ist die Menge der strukturell äquivalenten Muster einelementig.

Das Effizienzproblem kann man sehr wahrscheinlich lösen, wenn man zu einer Darstellung mit *Konstruktorvariablen* übergeht und die Konversionsrelation auf Typkonstruktoren mit Stelligkeit größer 0 erweitert: statt $C = \{\alpha \lhd set(\beta)\}$ aufzulösen in die beiden Restriktionsprobleme $C_1 = \{list(\alpha) \lhd set(\beta)\}$ und $C_2 = \{set(\alpha) \lhd set(\beta)\}$, ersetzt man C durch $C' = \{\kappa(\alpha) \lhd set(\alpha)\}$. Dies kann weiter vereinfacht werden, in das atomare Restriktionsproblem $C'' = \{\kappa \lhd set, \alpha \lhd \beta\}$, wenn man fordert, daß alle konvertierbaren Typkonstruktoren die gleichen Monotonieeigenschaften bzgl. der Konversionsrelation $\lhd$ besitzen. Also müssen beispielsweise alle Typkonstruktoren, die mit dem Funktionstyp in Relation stehen, im ersten Argument kovariant und im zweiten kontravariant sein.

Mit dieser Erweiterung könnte man dann auch auf den überladenen Applikationsoperator verzichten: man definiert $map(a, b) \lhd a' \to b' \longrightarrow\!\!\!\!\!\rightarrow\ a' \lhd a, b \lhd b', p_=(a)$ als Ersetzungsregel und erlaubt somit die Konversion von endlichen Abbildungen in Funktionen, vorausgesetzt für a ist der überladene Gleichheitsoperator definiert.

Allerdings enthält dieser Ansatz noch ein verstecktes Problem: ersetzt man in $\alpha \lhd \beta \to \gamma$ die Variable α durch den Konstrukturausdruck $\kappa(\alpha_1, \alpha_2)$ und transformiert das Restriktionsproblem in $\kappa \lhd \to, \beta \lhd \alpha_1, \alpha_2 \lhd \gamma$, dann geht die Information verloren, daß bei einer späteren Ersetzung von κ durch *map* zusätzlich die Restriktion $p_=(\alpha_1)$

betrachtet werden muß.

Eine Möglichkeit der Lösung dieses Problems bestünde darin, die Restriktion $\kappa(\alpha_1, \alpha_2) \lhd \beta \to \gamma$ in der betrachteten Restriktionsmenge C zu belassen, und die Restriktionen $\beta \lhd \alpha_1, \alpha_2 \lhd \gamma$ nur dann hinzuzufügen, wenn sie noch nicht von $C - \{\kappa(\alpha_1, \alpha_2) \lhd \beta \to \gamma\}$ impliziert werden. Die Restriktion $\kappa(\alpha_1, \alpha_2) \lhd \beta \to \gamma$ kann also erst dann aus C entfernt werden, wenn κ durch einen Typkonstruktor ersetzt wird. Wir verzichten auf eine formale Behandlung.

Noch eine abschließende Bemerkung zur Effizienz: da der Basisschritt der Restriktionsauflösung das Ersetzen einer Typvariablen durch ein strukturelles Muster ist, kann der bei Unifikationsalgorithmen übliche Trick des *structure sharing* nicht mehr eingesetzt werden. Somit gibt es Restriktionsprobleme, die nur mit exponentiellem Aufwand in atomar äquivalente umgewandelt werden können (siehe dazu die Diskussion in Abschnitt 6.4).

Kapitel 6

Konversionen und Überladungen

Im Folgenden untersuchen wir spezielle Aspekte der Konversionstypisierung und zeigen, daß sich Lösungen und allgemeinste Vereinfachungen für Restriktionsprobleme mit strukturellen Konversionen (Abschnitt 6.1) und parametrischen Überladungen (Abschnitt 6.2) bei geeigneten Einschränkungen an die Definition der Konversionsordnung und der Überladungsinstanzen effizient berechnen lassen. Anschließend zeigen wir, wie Konversionstypisierungen mit Überladungen durch Elimination unnötiger Restriktionen vereinfacht werden können (Abschnitt 6.3). Eine Diskussion der Ergebnisse und verwandter Arbeiten beschließt das Kapitel.

6.1 Erfüllbarkeit von Konversionsbedingungen

Nach Satz 5.18 läßt sich die Lösbarkeit der Konversionstypisierung zurückführen auf die Lösbarkeit eines Systems von Konversionsbedingungen über Variablen und Basistypen, sofern die Konversionsordnung strukturverträglich induktiv ist. Wir wollen nun untersuchen, unter welchen weiteren Voraussetzungen an die Konversionsordnung ein effizientes Verfahren zur Bestimmung der Lösbarkeit von Konversionsbedingungen angeben werden kann. Von besonderem Interesse ist dabei die Frage nach der Berechnung einer genauesten Vereinfachung.

Sei C eine Menge von Konversionsbedingungen der Form $a \lhd b$, mit $a, b \in F_0 \cup \mathcal{V}$, im Folgenden kurz Konversionsproblem genannt. Die Menge der in C vorkommenden Basistypen und Typvariablen bezeichnen wir mit $\mathit{atoms}(C)$, $\mathcal{V}(C)$ bezeichnet wie üblich die Menge der in C vorkommenden Typvariablen, und $\mathit{types}(C)$ sei definiert durch $\mathit{types}(C) = \mathit{atoms}(C) - \mathcal{V}(C)$.

Eine Normalform eines Konversionsproblems C gemäß Satz 5.18 läßt sich als gerichteten Graphen betrachten: Die Knoten des Graphen sind die in C vorkommenden

Variablen und Basistypen ($atoms(C)$), und für $a \lhd b \in C$ ist (a, b) eine Kante des Graphen. Der Graph zerfällt i.A. in mehrere Zusammenhangsgebiete, die durch die Äquivalenzklassen der Relation $\overset{C}{\sim}$ gegeben sind.

Lemma 6.1 (Letschert). *Unter der Annahme, daß $\lhd$ reflexiv ist, hat ein Konversionsproblem genau dann eine Lösung, wenn es eine Lösung gibt, die jeder Typvariablen einen Basistyp zuordnet.*

Beweis. Folgt aus der Tatsache, daß jede erfüllende Substitution allen Typvariablen strukturell äquivalente Typen zuordnen muß: Falls die Äquivalenzklasse einer Variablen α einen Basistyp b enthält, dann muß wegen $S(\alpha) \overset{se}{\sim} b$ auch $S(\alpha)$ ein Basistyp sein. Allen Variablen einer Äquivalenzklasse in der kein Basistyp vorkommt, kann aufgrund der Reflexivität der Konversionsordnung der gleiche Basistyp zugeordnet werden.[1] □

Korrollar 6.2. *Ein Konversionsproblem C über einer reflexiven Konversionsordnung ist entscheidbar, falls die Menge der Basistypen endlich ist; ist die Menge der Basistypen dagegen nur abzählbar, dann ist das Konversionsproblem i.A. nur semi-entscheidbar.*

Beweis. Die Semi-Entscheidbarkeit wird schon durch Satz 4.25 garantiert. Für endliche Basistypmengen ($|F_0| < \infty$) ist natürlich auch die Anzahl der Substitutionen endlich, die den Variablen $\{\alpha_1, \ldots, \alpha_n\}$ aus C Basistypen zuordnen. Nach Lemma 6.1 hat S genau dann eine Lösung, wenn eine dieser Substitutionen C erfüllt. □

Fügt man neben der Reflexivität als zusätzliche Anforderung an die Konversionsordnung die Transitivität hinzu, dann erhält man eine Quasiordnung. In einer solchen Konversionsordnung kann ein Typ t_1 sowohl Unter- als auch Obertyp eines Typs t_2 sein. Ein typische Konversionsordnung wäre dann z.B. $\{int \lhd real,\ real \lhd int\}$. In [FM88] wurde von Fuh und Mishra die Vermutung geäußert, daß ein polynomieller Lösbarkeitstest für solche Konversionsprobleme existiert. Wand und O'Keefe zeigten

[1] Das Resultat wurde erstmals in [Let86] publiziert. Eine ähnliche Aussage findet man auch in [WO89].

jedoch in [WO89], daß die Frage der Erfüllbarkeit für beliebige partielle Ordnungen NP-vollständig ist[2]. Der Beweis wird durch polynomielle Reduktion des 3-SAT Problems auf ein spezielles Konversionsproblem geführt und läßt sich auf Quasiordnungen übertragen, da partielle Ordnungen durch Hinzunahme des Antisymmetrieaxioms aus Quasiordnungen hervorgehen. Fügt man als zusätzliche Forderung an die Konversionsordnung noch hinzu, daß zu jeder Teilmenge $X \subseteq T_{F_0}$ eine kleinste obere bzw. größte untere Schranke existiert, dann kann man ein lineares Verfahren zur Überprüfung der Lösbarkeit angeben.

Die transitive Hülle des Graphen (C^*) beschreibt gerade die Menge aller von C implizierten Basistypkonversionen:

Proposition 6.3. *Für* $a, b \in T_{F_0}(\mathcal{V})$ *gilt:* $C \Vdash \{a \lhd b\} \iff (a, b) \in C^*$

Auch die Konversionsordnung auf den Basistypen kann als Graph repräsentiert werden, der ebenso wie ein Konversionsproblem in disjunkte Teile zerfällt.

$$K = \bigcup_{i=1..k} K_i$$

Ein Konversionsproblem C hat keine Lösung, falls C eine Konversion zwischen zwei Basistypen impliziert, die in getrennten Partitionen von K liegen. Diese Beobachtung kann man ausnutzen, um die Lösungssuche zu vereinfachen:

Die Antisymmetrie der Konversionsordnung erzwingt, daß jede erfüllende Substitution alle Variablen eines Zyklus identisch substituiert. Daher kann es keine Lösung des Konversionsproblems geben, falls C einen Zyklus mit mehr als einem Basistyp enthält.

Lemma 6.4. *Sei* $a \neq b$, $(a, b) \in C^*$ *und* $(b, a) \in C^*$, *dann gilt:*

$$(1)\ S \models C \Rightarrow S(a) = S(b)$$
$$(2)\ a \in F_0 \wedge b \in F_0 \Rightarrow \not\models C$$

Beweis. Wegen $a \lhd b \Rightarrow S(a) \lhd S(b)$ und $b \lhd a \Rightarrow S(b) \lhd S(a)$ folgt $S(a) = S(b)$ aus der Antisymmetrie der Konversionsordnung; (2) ist offensichtlich. □

[2] Zur Definition des Begriffs NP-vollständig siehe z.B. [GJ79]

Zur Elimination von Zyklen kann das bekannte Verfahren zur Berechnung von starken Zusammenhangskomponenten [AHU74, Abschnitt 5.5] eingesetzt werden, das sich in $O(\max(|Knoten|, |Kanten|))$ implementieren läßt. Das Verfahren berechnet die gröbste Partition $P = \biguplus P_i$ der Knoten eines Graphen, sodaß $v, w \in P_i \Rightarrow \{(v,w),(w,v)\} \subseteq C^*$. Falls eines der P_i mehr als einen Basistyp enthält, dann hat C keine Lösung. Ansonsten können wir aus jedem der P_i einen Repräsentanten $r(P_i)$ auswählen, alle Variablen von P_i in C durch $r(P_i)$ ersetzen und das resultierende Problem vereinfachen. Dabei sei

$$r(P_i) \stackrel{\text{def}}{=} \begin{cases} b & \text{falls } P_i \text{ ein } b \in F_0 \text{ enthält,} \\ \alpha & \text{für irgendein } \alpha \in P_i. \end{cases}$$

Die Überprüfung, ob eine Zusammenhangskomponente P_i mehr als einen Basistyp enthält, läßt sich auf einfache Weise in das Verfahren integrieren und ändert nichts an der Zeitschranke.

Satz 6.5. *Sei S die Substitution, die jedem $\alpha \in \mathcal{V}(C)$ den Repräsentanten der α umfassenden starken Zusammenhangskomponente zuordnet und C' das Konversionsproblem, das man durch Anwenden von S auf C und Elimination von Gleichungen der Form $a \lhd a$ erhält. Dann gilt:*

1. *S ist eine vereinfachende Substitution von C,*
2. *C' ist azyklisch.*

Beweis. (1) folgt aus Lemma 6.4, (2) ist offensichtlich. □

Die Antisymmetrie ist allerdings keine hinreichende Bedingung für die effiziente Konstruktion einer Lösung. Es kann dennoch exponentiell viele Lösungen geben. Beispiel:

$$C_n = \{\, int \lhd \alpha_i \mid i = 1..n \,\} \cup \{\, \alpha_i \lhd real \mid i = 1..n \,\}$$

Hier ist jede Substitution S mit $S(\alpha_i) \in \{int, real\}$ eine Lösung; somit gibt es 2^n Lösungen für C.

Lemma 6.6. *Sei C ein azyklisches Konversionsproblem mit $types(C) = \emptyset$. Dann ist die identische Substitution eine genaueste Vereinfachung von C.*

Beweis. Es ist klar, daß die Identität eine vereinfachende Substitution für C ist, nach Lemma 4.33 verbleibt somit zu zeigen, daß:

$$(a)\ \forall \alpha, \beta \in \mathcal{V}(C) : \exists L \models C : L(\alpha) \neq L(\beta)$$
$$(b)\ \forall \alpha \in \mathcal{V}(C) : \exists L_1, L_2 \models C : \mathit{root}(L_1(\alpha)) \neq \mathit{root}(L_2(\alpha))$$

(a) C enthält mindestens 2 Typvariablen α, β für die o.B.d.A. $(\alpha, \beta) \in C^*$ oder $(\beta, \alpha) \notin C^*$. Sei $b_1 \lhd b_2$ eine gültige Konversion zwischen $b_1, b_2 \in F_0$. Die Substitution

$$L(\gamma) \stackrel{\text{def}}{=} \begin{cases} b_1 & \text{falls } (\gamma, \alpha) \in C^*, \\ b_2 & \text{sonst} \end{cases}$$

ist eine Lösung für C: Eine Konversionsbedingung $\beta \lhd \beta'$ ist nur dann nicht erfüllt, wenn $L(\beta) = b_2$ und $L(\beta') = b_1$. Aus der Definition von L folgt $(\beta', \alpha) \in C^*$, was zusammen mit $\beta \lhd \beta'$ auch $(\beta, \alpha) \in C^*$ impliziert. Dies steht im Widerspruch zur Zyklenfreiheit von C.

(b) Für jeden Typ $t \in T_F$ ist die Substitution $L_t = \{\alpha_i \mapsto t\}$ eine Lösung von C. Da o.B.d.A. $|T_F| > 1$, existieren L_{t_1}, L_{t_2} mit $\mathit{root}(L_{t_1}(\alpha)) = \mathit{root}(t_1) \neq \mathit{root}(t_2) = \mathit{root}(L_{t_2}(\alpha))$. □

Von Reynolds [Rey85] und später von Letschert [Let86] wurde von der Konversionsordnung verlangt, daß zu je zwei Typen t_1, t_2 entweder überhaupt kein gemeinsamer Obertyp (bzw. Untertyp) existiert, oder aber ein kleinster (bzw. größter), die mit $t_1 \sqcup t_2$ (bzw. $t_1 \sqcap t_2$) bezeichnet werden. Mit anderen Worten: die Konversionsordnung bildet einen Verband, bzw. eine endliche Vereinigung disjunkter Verbände.

Diese Eigenschaft der Konversionsordnung legt folgendes Verfahren nahe: Berechne für jede Variable α die Menge der unteren Schranken in C

$$L_C(\alpha) = \{\, b \in F_0 \mid b \lhd \alpha \in C^* \,\}$$

Falls $L_C(\alpha)$ keine kleinste obere Schranke besitzt, dann gibt es keine Lösung des Konversionsproblems. Ansonsten kann man die Substitution S_1 mit

$$S_1(\alpha) = \bigsqcup L_C(\alpha)$$

auf das Konversionsproblem anwenden und für die verbleibende Menge von Variablen die Menge der oberen Schranken in $S_1(C)$ berechnen.

$$U_C(\alpha) = \{\, b \in F_0 \mid \alpha \lhd b \in C^* \,\}$$

Hat $U_{S_1(C)}$ keine größte untere Schranke, dann schlägt die Überprüfung fehl. Im anderen Fall ist $S_2 \circ S_1$ mit

$$S_2(\alpha) = \bigsqcap U_C(\alpha)$$

eine Lösung des Konversionsproblems.

Das Verfahren ist zwar wohldefiniert, aber nicht vollständig, d.h., es schlägt manchmal fehl, obwohl eine Lösung existiert: Für das Konversionsproblem

$$\{\, b_1 \lhd \alpha,\ \alpha \lhd b, \beta \lhd b_2, \beta \lhd \alpha \,\}$$

mit der Konversionsordnung

$$\{\, b_1 \lhd b, b_2 \lhd b \,\}$$

ist $S_1(\alpha) = b_1$. Der aus der Anwendung von S_1 resultierende Konversionsgraph

$$\{\, b_1 \lhd b,\ \beta \lhd b_1, \beta \lhd b_2 \,\}$$

besitzt keine Lösung, da b_1 und b keine gemeinsame untere Schranke besitzen. Das Konversionsproblem ist allerdings lösbar mit der Substitution

$$\{\, \alpha \mapsto b,\ \beta \mapsto b_2 \,\}$$

Das Problem liegt darin, daß die möglichen Belegungen von β sowohl durch $U_C(\beta)$ als auch durch die Wahl von $S_1(\alpha)$ eingeschränkt werden, was bei der Konstruktion von S_1 aber nicht berücksichtigt wird. Es ist klar, daß auch eine Umkehrung der Vorgehensweise, also Beginn mit den maximalen Elementen der Konversionsordnung, nicht weiterhilft. Fordert man jedoch, daß jede Zusammenhangskomponente der Konversionsordnung einen Verband bildet, dann existieren zwei ausgezeichnete, wohldefinierte Substitutionen $S_\sqcup$ und $S_\sqcap$ mit

$$\begin{aligned} S_\sqcup(\alpha) &= \bigsqcup L_C(\alpha) \\ S_\sqcap(\alpha) &= \bigsqcap U_C(\alpha) \end{aligned}$$

und es gilt

Satz 6.7. *Ein azyklisches Konversionsproblem mit types*$(C) \subseteq K_k$ *hat genau dann eine Lösung, wenn für alle* $a \lhd b$ *aus* C *die Ungleichung* $S_\sqcup(a) \lhd S_\sqcap(b)$ *erfüllt ist. In diesem Fall gilt* $S_\sqcup \models C$ *und* $S_\sqcap \models C$*. Darüber hinaus ist die Substitution* S_g *mit*

$$S_g(\alpha) \stackrel{\text{def}}{=} \begin{cases} b & \textit{falls } b = S_\sqcup(\alpha) = S_\sqcap(\alpha), \\ \alpha & \textit{sonst} \end{cases}$$

eine genaueste Vereinfachung von C*.*

Beweis. Man beachte zunächst, daß $\bigsqcup \emptyset = \bot_k$ und $\bigsqcap \emptyset = \top_k$, wobei $\bot_k$ das kleinste und $\top_k$ das größte Element in K_k sind. Da K_k ein Verband ist, existieren Supremum und Infimum für beliebige Teilmengen $X \subseteq K_k$, sodaß $S_\sqcup$ bzw. $S_\sqcap$ wohldefiniert sind.

Sei L eine Lösung für C. Wir zeigen, daß

$$\begin{aligned} &(1)\ S_\sqcup(a) \lhd S_\sqcup(b)\,, \\ &(2)\ S_\sqcap(a) \lhd S_\sqcap(b) \text{ und} \\ &(3)\ S_\sqcup(a) \lhd S_\sqcap(b) \end{aligned}$$

für alle $a \lhd b \in C$ gültige Konversionen sind. Dazu unterscheiden wir vier Fälle:

$a \in F_0,\ b \in F_0$: Dann ist $a \lhd b$ nach Voraussetzung eine gültige Konversion, da L sonst keine Lösung wäre. Offensichtlich gelten wg. $S(a) = a$ und $S(b) = b$ die Gleichungen 1–3.

$a \in F_0,\ b \in \mathcal{V}$: Wegen $a \in L_C(b)$ und $S(a) = a$ und $x \lhd x \sqcup y$ gilt $a \lhd \bigsqcup L_C(b) = S_\sqcup(b)$ und damit (1). Da $a \lhd L(b)$ eine gültige Konversion ist, muß $L(b) \lhd \bigsqcap U_C(b)$ gelten und damit folgen (2) und (3) aus der Transitivität von $\lhd$.

$a \in \mathcal{V},\ b \in F_0$: Wegen $b \in U_C(a)$ und $x \sqcap y \lhd x$ gilt $S_\sqcup(a) = \bigsqcap U_C(a) \lhd b = S_\sqcap(b) = S_\sqcup(b)$ und damit (2) und (3). Da $L(a) \lhd b$ eine gültige Konversion ist, muß $\bigsqcup L_C(a) \lhd L(a)$ gelten und damit folgt (1) aus der Transitivität von $\lhd$.

$a \in \mathcal{V},\ b \in \mathcal{V}$: Aus $a \lhd b \in C$ folgt $L_C(a) \subseteq L_C(b)$ bzw. $U_C(b) \subseteq U_C(a)$ und damit auch $\bigsqcup L_C(a) \lhd \bigsqcup L_C(b)$ bzw. $\bigsqcap U_C(b) \lhd \bigsqcap U_C(a)$, sodaß (1) und (2) erfüllt sind. Da $L(a) \lhd L(b)$ eine gültige Konversion ist, muß sowohl $S_\sqcup(a) = \bigsqcup L_C(a) \lhd L(a)$ als auch $L(b) \lhd \bigsqcap U_C(a) = S_\sqcap(a)$ gelten. Wegen $L(a) \lhd L(b)$ folgt $S_\sqcup(a) \lhd S_\sqcap(b)$ und damit (3).

Falls $S_\sqcup(a) \lhd S_\sqcap(b)$ für alle $a \lhd b \in C$, dann sind sowohl $S_\sqcup$ als auch $S_\sqcap$ Lösungen von C. S_g ist offensichtlich eine Vereinfachung von C, aus Lemma 4.33 folgt, daß $S_g(C)$ nicht weiter vereinfacht werden kann. □

Die Menge der oberen bzw. unteren Schranken muß wegen der Zyklenfreiheit von C nicht wirklich berechnet werden: man berechnet in einem ersten Schritt eine topologische Sortierung der Knoten des Graphen. Die Substitutionen $S_\sqcup$ und $S_\sqcap$ können dann in zwei Durchgängen über die Sortierung berechnet werden.

Definition 6.8. Eine topologische Sortierung eines Graphen $G = (V, E)$ ist eine bijektive Abbildung $\pi : V \to 1..|V|$ derart, daß für alle $(v, w) \in E$ die Ungleichung $\pi(v) < \pi(w)$ gilt.

Bekanntermaßen[3] hat ein Graph genau dann eine topologische Sortierung, wenn er azyklisch ist. Eine topologische Sortierung kann in $O(|V| + |E|)$ Zeiteinheiten berechnet werden.

Algorithmus 6.1. Berechnung von $S_\sqcup, S_\sqcap$. Sei C ein atomares Konversionsproblem mit $types(C) \subseteq K$ und $n = |\mathcal{V}(C)|$.

for $i := 1$ **to** n **do**
  $S_\sqcup(\pi^{-1}(i)) := \bigsqcup \{ S_\sqcup(a) \mid (a, \pi^{-1}(i)) \in C \}$
for $i := n$ **downto** 1 **do**
  $S_\sqcap(\pi^{-1}(i)) := \bigsqcap \{ S_\sqcap(a) \mid (\pi^{-1}(i), a) \in C \}$

Satz 6.9. *Algorithmus 6.1 berechnet $S_\sqcup$ und $S_\sqcap$ korrekt in $O(\mathcal{V}(C) + |C|)$ Schritten.*

Beweis. Die Zeitabschätzung ergibt sich aus der Tatsache, daß für jede Variable α alle direkten Vorgänger in C zur Berechnung von $S_\sqcup(\alpha)$ bzw. alle Nachfolger zur

[3] Siehe z.B. [Meh84].

Berechnung von $S_\sqcap(\alpha)$ inspiziert werden müssen. Zum Beweis der Korrektheit zeigt man die Schleifeninvarianten

$$\begin{array}{ll} (1) & \forall j < i : S_\sqcup(\pi^{-1}(j)) = \bigsqcup L_C(\pi^{-1}(j)) \\ (2) & \forall j > i : S_\sqcap(\pi^{-1}(j)) = \bigsqcap U_C(\pi^{-1}(j)) \end{array}$$

(1) Für $i = 1$ ist die Behauptung offensichtlich erfüllt. Sei also $i > 1$ und (1) gelte für $j < i$. Sei $\alpha = \pi^{-1}(j)$.

$$\begin{aligned} S_\sqcup(\alpha) &= \bigsqcup \{\, S_\sqcup(a) \mid a \lhd \alpha \in C \,\} \\ &= \bigsqcup \{\, \bigsqcup L_C(a) \mid a \lhd \alpha \in C \,\} \qquad (*) \\ &= \bigsqcup \bigcup_{a \lhd \alpha \in C} L_C(a) \\ &= \bigsqcup L_C(\alpha) \end{aligned}$$

Gleichung (*) folgt aus $\pi(a) < \pi(\alpha)$ und der Induktionshypothese.

(2) Analog (1), wobei $i = n$ die Induktionsverankerung ist, und der Induktionsschritt von $j > i$ nach i erfolgt:

$$\begin{aligned} S_\sqcap(\alpha) &= \bigsqcap \{\, S_\sqcap(a) \mid \alpha \lhd a \in C \,\} \\ &= \bigsqcap \{\, \bigsqcap U_C(a) \mid \alpha \lhd a \in C \,\} \\ &= \bigsqcap \bigcup_{\alpha \lhd a \in C} U_C(a) \\ &= \bigsqcap U_C(\alpha) \end{aligned}$$

□

Die Bedingung $S_\sqcup(\alpha) \lhd S_\sqcap(\alpha)$ kann anschließend mit linearem Aufwand in der Anzahl der Konversionsbedingungen überprüft werden. Insgesamt folgt daraus:

Satz 6.10. *Ein Konversionsproblem $C \subseteq \mathcal{V} \cup F_o$ ist in $O(max(n, m))$ entscheidbar, falls die Konversionsordnung auf Basistypen in disjunkte Verbände zerfällt. Falls eine Lösung existiert, dann gibt es auch eine genaueste Vereinfachung.*

Wir erhalten den folgenden Algorithmus:

Algorithmus 6.2. Unterteile das Konversionsproblem C in seine Zusammenhangskomponenten $C_1, \ldots, C_n$ und führe für jedes der C_i die nachfolgenden Schritte aus.

1. Eliminiere Zyklen in C_i.
2. Überprüfe ob ein k existiert mit $types(C_i) \subseteq K_k$.
3. Berechne $S_\sqcap(C_i)$, $S_\sqcup(C_i)$
4. Überprüfe ob $S_\sqcup(a) \lhd S_\sqcap(b)$ für alle $a \lhd b \in C_i$ eine gültige Konversion ist.
5. Berechne $S_g(C_i)$ und eliminiere redundante Gleichungen.
6. Berechne die transitive Reduktion des Ergebnisgraphen $S_g(C_i)^{red}$.

Die Substitution $S_g \circ S_c$ ist eine genaueste Vereinfachung für C.

Dieses Verfahren kann mit einem Aufwand von $O(max(n * e_{red}, n, e))$ implementiert werden, wobei e_{red} die Anzahl der Kanten des transitiv reduzierten Graphen ist.[4][5]

Die Forderung nach dem Zerfall der Konversionsordnung in eine Menge disjunkter Verbände scheint auf den ersten Blick eine starke Einschränkung zu sein. Die Erfahrungen mit der Entwicklung des SAMPλE-Typsystems haben gezeigt, daß man durchaus sinnvolle Konversionsordnungen finden kann, die dieser Einschränkung genügen. Beispielsweise ist es möglich alle Zahltypen der Sprache in einer Kette anzuordnen

$$nat \lhd int \lhd rat \lhd real \lhd double \lhd complex$$

und sonst gar keine Basistypkonversionen zu erlauben.

6.2 Konversionen und parametrische Überladungen

Wenden wir uns nun dem Problem der Lösbarkeit zyklenfreier Restriktionsmengen zu, die neben strukturellen Konversionsbedingungen auch parametrische Überladungsrestriktionen beinhalten. Wir geben Bedingungen an, unter denen sich die Lösbarkeit eines Restriktionsproblems und eine genaueste Vereinfachung des Problems effizient berechnen läßt.

[4] Siehe z.B. [Sim89] für einen linearen Algorithmus zur Berechnung einer minimalen transitiven Reduktion.

[5] In der aktuellen SAMPλE-Implementierung sind alle beschriebenen Schritte direkt in den Inferenzalgorithmus integriert, um mögliche Typfehler in Teilausdrücken sofort melden zu können.

Die einfachste Möglichkeit die Lösbarkeit von Konversionsmengen mit überladungssortierten Typvariablen zu garantieren, ist die Forderung, daß für jede Teilmenge von überladenen Operatoren X entweder $\{b \in K \mid X{\uparrow}b\} = \emptyset$ oder $\{b \in K \mid X{\uparrow}b\} = K$ gelten soll.[6] In Worten: entweder ein überladener Operator ist für keinen Basistyp einer starken Zusammenhangskomponente der Konversionsordnung definiert, oder aber auf allen Typen der Zusammenhangskomponente. Äquivalent dazu ist die Forderung

$$\forall o : \{b \in K \mid o{\uparrow}b\} = \emptyset \vee \{b \in K \mid o{\uparrow}b\} = K$$

In diesem Fall beantwortet das Verfahren des letzten Abschnitts schon die Frage der Lösbarkeit und liefert außerdem die genaueste Vereinfachung des Konversionsproblems. Leider erfüllt das SAMPλE-Typsystem diese Bedingung nicht, da *int* und *real* eine Zusammenhangskomponente bilden und beispielsweise die Operatoren *pred* und *succ* nur für *int* definiert sind und der Operator / nur für *real* zulässig ist (siehe Seite 43).

Weniger restriktiv ist dagegen die Forderung

$$\forall K.\forall b \in K.\forall X. \begin{cases} X{\uparrow}_K \cap \mathcal{A}_K(b) \neq \emptyset \implies \bigsqcup (X{\uparrow}_K \cap \mathcal{A}_K(b)) \in X{\uparrow}_K \\ X{\uparrow}_K \cap \mathcal{B}_K(b) \neq \emptyset \implies \bigsqcap (X{\uparrow}_K \cap \mathcal{B}_K(b)) \in X{\uparrow}_K \end{cases} \tag{6.1}$$

wobei

$$\begin{aligned} X{\uparrow}_K &= \{b \in K \mid X{\uparrow}b\}\,,^{7} \\ \mathcal{A}_C(a) &= \{b \mid (a,b) \in C^*\}\,,^{8} \\ \mathcal{B}_C(a) &= \{b \mid (b,a) \in C^*\}\,. \end{aligned}$$

In Worten: für jeden Basistyp b einer starken Zusammenhangskomponente, und für jede Teilmenge überladener Operatoren X sind die folgenden Bedingungen erfüllt.

a) Entweder ist kein Operator für einen Basistyp, der in der Konversionsordnung oberhalb von b liegt, definiert, oder die Menge der Basistypen oberhalb von b, für die alle Operatoren in X definiert sind, besitzt eine kleinste untere Schranke

[6] Zur Vereinfachung der Notation schreiben wir $X{\uparrow}\tau$ falls $\{p(\tau) \mid p \in X\}$ erfüllbar ist und α_X für $X = \{p \mid p(\alpha) \in C\}$ bei bekanntem C.

[8] Man beachte, daß $\emptyset{\uparrow}_K = K$ gilt, sodaß für $X = \emptyset$ Gleichung (6.1) immer erfüllt ist.

[8] $\mathcal{A}$ steht für *above*, $\mathcal{B}$ steht für *below*.

gemäß Konversionsordnung, für die ebenfalls alle Operatoren in X definiert sind.

b) Entweder ist kein Operator für einen Basistyp, der in der Konversionsordnung unterhalb von b liegt, definiert, oder die Menge der Basistypen unterhalb von b, für die alle Operatoren in X definiert sind, besitzt eine größte obere Schranke gemäß Konversionsordnung, für die alle Operatoren in X definiert sind.

Für eine Sprache mit laufzeitauflösbaren Überladungen und Konversionen, wie z.B. SAMPλE, ist leicht einzusehen, warum dies eine notwendige Forderung ist. Angenommen ein Operator o mit Überladungsschema $\$ \to \tau$ wird auf einen Wert c vom Typ b angewendet, dann muß anhand des Typs b entschieden werden, welche Überladungsinstanz von o zur Berechnung von $o(c)$ eingesetzt wird. Eine eindeutige Auswahl dieser Instanz ist nur dann möglich, wenn in der Konversionsordnung der Basistypen ein kleinstes Element existiert, für das o eine Überladungsinstanz besitzt.

Für Sprachen, deren Semantik über eine Übersetzung in eine rein parametrisch polymorphe Zwischensprache definiert wird, scheint die Forderung zumindest sinnvoll, da sie die manuelle Herleitung von Typisierungen erleichtert und somit für den Programmierer leichter nachzuvollziehen ist, allerdings scheint sie nicht zwingend erforderlich zu sein.

Satz 6.11. *Sei C ein Restriktionsproblem mit types$(C) \subseteq K$ und $S^o_\sqcap$, $S^o_\sqcup$ definiert durch*

$$S^o_\sqcap(\alpha_X) = \bigsqcap (X{\uparrow_K} \cap \mathcal{B}_K(\bigsqcap \{S^o_\sqcap(b) \mid b \in (\mathcal{A}_C(\alpha_X) - \{\alpha_X\})\}))$$
$$S^o_\sqcup(\alpha_X) = \bigsqcup (X{\uparrow_K} \cap \mathcal{A}_K(\bigsqcup \{S^o_\sqcup(a) \mid a \in (\mathcal{B}_C(\alpha_X) - \{\alpha_X\})\}))$$

und S_g definiert wie in Satz 6.7 und die Konversionsordnung auf den Basistypen erfülle Gleichung (6.1). C hat genau dann eine Lösung, wenn für alle $a \lhd b$ aus C die Ungleichung $S^o_\sqcup(a) \lhd S^o_\sqcap(b)$ erfüllt ist und $X{\uparrow}S^o_\sqcup(\alpha_X) \wedge X{\uparrow}S^o_\sqcap(\alpha_X)$ für alle $\alpha \in \mathcal{V}(C)$ gilt. In diesem Fall gilt $S^o_\sqcup \models C$ und $S^o_\sqcap \models C$. Die Substitution S_g ist eine genaueste Vereinfachung von C.

Beweis. Daß $S^o_\sqcup$ und $S^o_\sqcap$ wohldefiniert sind, folgt durch noethersche Induktion über die Ordnung $a \leq b \iff a \lhd b \in (C^* \cap \mathcal{V}^2)$ für $S^o_\sqcap$, bzw. $\geq$ für $S^o_\sqcup$, unter Verwendung

von Bedingung (6.1). Auf die gleiche Weise zeigt man, daß für jede Lösung $L \models C$, $L(\alpha) \lhd S^o_{\sqcup}(\alpha)$ (noethersche Induktion über $\leq$) und $S^o_{\sqcap}(\alpha) \lhd L(\alpha)$ für alle $\alpha \in \mathcal{V}(C)$ gilt (noethersche Induktion über $\geq$).

Sei L eine Lösung für C. Wir zeigen, daß

$$\begin{aligned}&(1)\ S^o_{\sqcup}(a) \lhd S^o_{\sqcup}(b)\,,\\ &(2)\ S^o_{\sqcap}(a) \lhd S^o_{\sqcap}(b) \text{ und}\\ &(3)\ S^o_{\sqcup}(a) \lhd S^o_{\sqcap}(b)\end{aligned}$$

für alle $a \lhd b \in C$ gültige Konversionen sind. Dazu unterscheiden wir vier Fälle:

$a \in F_0,\ b \in F_0$: Dann ist $a \lhd b$ nach Voraussetzung eine gültige Konversion, da L sonst keine Lösung wäre. Offensichtlich gelten wg. $S(a) = a$ und $S(b) = b$ die Gleichungen 1–3.

$a \in F_0,\ b \in \mathcal{V}$: Wegen $a \in \mathcal{B}_C(b) - \{b\}$ und $S(a) = a$ und $x \lhd x \sqcup y$ gilt $a \lhd D(b)$, wobei $D(b) = \bigsqcup \{S^o_{\sqcup}(b') \mid b' \in \mathcal{B}_C(b) - \{b\}\}$; und da $D(b) \lhd \bigsqcup (X\uparrow_K \cap \mathcal{A}_K(D(b)))$ gilt, folgt (1) aus der Transitivität von $\lhd$.

Da $a \lhd L(b)$ eine gültige Konversion ist, und wie o.a. $L(b) \lhd S^o_{\sqcap}(b)$ gelten muß, folgen (2) und (3) aus der Transitivität von $\lhd$.

$a \in \mathcal{V},\ b \in F_0$: Wegen $b \in \mathcal{A}_C(a) - \{a\}$ und $x \sqcap y \lhd x$ gilt $E(a) \lhd b = S^o_{\sqcap}(b) = S^o_{\sqcup}(b)$, wobei $E(a) = \bigsqcap \{S^o_{\sqcap}(a') \mid a' \in \mathcal{A}_C(a) - \{a\}\}$. Wegen $\bigsqcap (X\uparrow_K \cap \mathcal{B}_K(E(a))) \lhd E(a)$ folgt (2) und (3).

Da $L(a) \lhd b$ eine gültige Konversion ist, und wie o.a. $S^o_{\sqcup} \lhd L(a)$ gelten muß, folgt (1) aus der Transitivität von $\lhd$.

$a \in \mathcal{V},\ b \in \mathcal{V}$: Aus $a \lhd b \in C$ und der Definition von $S^o_{\sqcup}$ folgt $S^o_{\sqcup}(a) \in \{S^o_{\sqcup}(b') \mid b' \in \mathcal{B}_C(b) - \{b\}\}$ und damit $S^o_{\sqcup}(a) \lhd D(b)$ und wegen $D(b) \lhd \bigsqcup (X\uparrow_K \cap \mathcal{A}_K(D(b))) = S^o_{\sqcup}(b)$ folgt (1).

Analog folgt (2): Aus $a \lhd b \in C$ und der Definition von $S^o_{\sqcap}$ folgt $S^o_{\sqcap}(b) \in \{S^o_{\sqcap}(a') \mid a' \in \mathcal{A}_C(a) - \{a\}\}$ und damit $E(a) \lhd S^o_{\sqcup}(b)$ und wegen $\bigsqcap (X\uparrow_K \cap \mathcal{B}_K(E(b))) \lhd E(b)$ folgt $S^o_{\sqcap}(a) \lhd S^o_{\sqcap}(b)$.

Da $L(a) \lhd L(b)$ eine gültige Konversion ist, muß sowohl $S^o_\sqcup(a) \lhd L(a)$ als auch $L(b) \lhd S^o_\sqcap(b)$ gelten. Wegen $L(a) \lhd L(b)$ folgt $S^o_\sqcup(a) \lhd S^o_\sqcap(b)$ und damit (3).

Falls $S^o_\sqcup(a) \lhd S^o_\sqcap(b)$ für alle $a \lhd b \in C$, dann sind sowohl $S^o_\sqcup$ als auch $S^o_\sqcap$ Lösungen von C. S_g ist offensichtlich eine Vereinfachung von C, aus Lemma 4.33 folgt, daß $S_g(C)$ nicht weiter vereinfacht werden kann. □

Beispiel: die in Abbildung 3.3 auf Seite 43 angegebene Überladungsannahme der vordefinierten SAMPλE-Operatoren erfüllt mit der Basistypkonversionsordnung $\mathit{int} \lhd \mathit{real}$ die Bedingung (6.1). Allgemein gilt:

Proposition 6.12. *Falls jede Zusammenhangskomponente der Konversionsordnung eine Kette bildet, ist Bedingung (6.1) für beliebige Überladungsannahmen erfüllt.*

Beweis. Folgt aus der Tatsache, daß jede Teilmenge einer Kette wiederum eine Kette ist und somit einen Verband bildet. Daher besitzt jede Teilmenge $B \subseteq K$ ein kleinstes und größtes Element in B. □

Die Berechnung der Substitutionen $S^o_\sqcup$ und $S^o_\sqcap$ erfolgt analog Algorithmus 6.1 unter Ausnutzung der topologischen Sortierbarkeit des Restriktionsproblems C.

Algorithmus 6.3. Berechnung von $S^o_\sqcup$ und $S^o_\sqcap$. Sei $n = |\mathcal{V}(C)|$ und π eine topologische Sortierung von C.

for $i := 1$ **to** n **do**
  $S^o_\sqcup(\pi^{-1}(i)) := \bigsqcup (X\!\uparrow_K \cap\, \mathcal{A}_K(\bigsqcup \{\, S^o_\sqcup(a_X) \mid (a_X, \pi^{-1}(i)) \in C \,\}))$
for $i := n$ **downto** 1 **do**
  $S^o_\sqcap(\pi^{-1}(i)) := \bigsqcap (X\!\uparrow_K \cap\, \mathcal{B}_K(\bigsqcap \{\, S^o_\sqcap(a_X) \mid (\pi^{-1}(i), a_X) \in C \,\}))$

Satz 6.13. *Algorithmus 6.3 berechnet $S^o_\sqcup$ und $S^o_\sqcap$ korrekt in $O(\mathcal{V}(C)+|C|)$ Schritten.*

Beweis. Man beachte, daß sowohl $\mathcal{A}_K(b)$ und $\mathcal{B}_K(b)$ als auch $X\!\uparrow_K$ nicht von dem konkreten Restriktionsproblem abhängen. Damit kann auch die Schnittmenge $X\!\uparrow_K \cap\, \mathcal{A}_K(b)$ bzw. $X\!\uparrow_K \cap \mathcal{B}_K(b)$ vorab berechnet werden. Somit gilt die gleiche Zeitabschätzung wie für Algorithmus 6.1. Die Korrektheit folgt durch Induktionsbeweis analog zu Satz 6.9. □

Für Restriktionsprobleme, die nur Typvariablen enthalten, kann man aufgrund des DEXPTIME-Resultats für parametrische Überladungen im Allgemeinen keinen effizienten Algorithmus erwarten. Fordert man jedoch, daß lösbare Restriktionspobleme immer Basistyplösungen besitzen, kann man den obigen Algorithmus für jede Zusammenhangskomponente anwenden, um die Lösbarkeit zu überprüfen. Das SAMPλE-System erfüllt diese Anforderung.

6.3 Vereinfachung von Konversionstypaussagen

Konversionstypisierungen, wie sie von Algorithmus $\mathcal{D}$ berechnet werden, können recht komplex werden. Wie der Einsatz einer ersten Version der SAMPλE-Umgebung mit Konversionstypinferenz gezeigt hat, treten in der praktischen Anwendung Restriktionsmengen auf, deren Größe linear mit der Größe let-freier Lambdaterme wächst. Wir stellen nun kurz eine Methode von Fuh und Mishra vor [FM89], mit deren Hilfe die Restriktionsmengen in Konversionstypisierungen verkleinert werden können, und zeigen dann, wie man die Methode erweitert, um parametrische Überladungen behandeln zu können.[9] Das auf parametrische Überladungen erweiterte Verfahren wurde im SAMPλE-System implementiert. Dabei hat sich gezeigt, daß die Methode sehr effizient ist: in der Regel werden Konversionsrestriktionsmengen vollständig eliminiert oder auf wenige Restriktionen reduziert.

Grundlegend für die Entwicklung der Vereinfachungsverfahren ist die Definition einer „stärkeren“ Instanzrelation, die mehr Typaussagen in Relation setzt, als die „schwächere“ Instanzrelation von Mitchell [Mit84].

Definition 6.14. Eine Typaussage $C', \Gamma \vdash M : \tau'$ ist eine *verzögerte Instanz*[10] von $C, \Gamma \vdash M : \tau$, falls die folgenden Bedingungen erfüllt sind:

$$(1)\ C' \Vdash S(C)$$
$$(2)\ C' \Vdash S(\tau) \lhd \tau'$$
$$(3)\ C' \Vdash \Gamma'(x) \lhd S(\Gamma(x)) \quad \forall x \in \mathcal{FV}(M)$$

[9] Man beachte, daß durch die Verkleinerung der Restriktionsmengen auch sämtliche bisher vorgestellten Algorithmen zur Berechnung von Lösungen bzw. genauester Vereinfachungen effizienter ablaufen.

[10] engl. „lazy instance“.

Bedingung (1) ist in beiden Instanzrelationen gleich, Bedingung (2) ersetzt die Bedingung $S(\tau) = \tau'$ und Bedingung (3) ersetzt $\Gamma' = S(\Gamma)$.

Für den praktischen Einsatz bedeutet die verzögerte Instanzrelation, daß die von Algorithmus $\mathcal{D}$ berechnete Typaussage $C, \Gamma \vdash M : \tau$ vereinfacht werden kann, falls eine Substitution S existiert, welche die Bedingungen der verzögerten Instanzrelation erfüllt. Wie in [FM89] gezeigt, genügt es, Substitutionen zu betrachten, die Typvariablen durch atomare Typen ersetzen. Da $\mathcal{V}(C)$ endlich ist, kann man sukzessive alle Substitutionen aufzählen, die keine reinen Variablenumbenennungen sind und Variablen aus $\mathcal{V}(C)$ auf Basistypen oder andere Variablen aus $\mathcal{V}(C)$ abbilden, die Bedingungen 1–3 prüfen, die Substitution anwenden und die Restriktionsmenge anschließend gemäß der Implikationsrelation vereinfachen. Da jedesmal mindestens eine Variable ersetzt wird, muß dieser Prozeß terminieren, sodaß man eine minimale Typisierung erhält.

Bedauerlicherweise erfordert die Aufzählung der Substitutionen exponentiellen Aufwand, sodaß dieses Verfahren für den praktischen Einsatz untauglich erscheint. Fuh und Mishra beobachteten jedoch, daß die Mehrzahl der vereinfachenden Substitution nur einzelne Variablen involvieren, die man außerdem noch relativ einfach bestimmen kann.

Dazu werden von Fuh und Mishra zwei Methoden definiert: die erste ersetzt nur *interne* Typvariablen, d.h. solche Variablen, die weder in τ, noch in Γ frei vorkommen, die zweite ersetzt *externe* Variablen, d.h. solche, die in τ und Γ vorkommen.

Wir betrachten zuerst die Ersetzung interner Typvariablen.

Definition 6.15. Ein atomarer Typ $r \in \mathit{atoms}(C)$ G-subsumiert $\alpha \in \mathcal{V}(C)$, geschrieben $\alpha \leq_G r$, falls die folgenden Bedingungen erfüllt sind:

$$(a)\ \mathcal{A}_C(\alpha) - \{\alpha\} \subseteq \mathcal{A}_C(r),$$
$$(b)\ \mathcal{B}_C(\alpha) - \{\alpha\} \subseteq \mathcal{B}_C(r).$$

Darauf aufbauend, wird eine Transformationsrelation $\mapsto_G$ auf Typisierungen definiert: $C, \Gamma \vdash M : \tau \mapsto_G SC, \Gamma \vdash \tau$, falls $S = \{\alpha \mapsto r\}$ für $\alpha \leq_G r$ und $\alpha \notin \mathcal{FV}(\Gamma) \cup \mathcal{V}(\tau)$. Fuh und Mishra zeigen, daß $\mapsto_G$ eine Funktion auf Typisierungen definiert, sodaß jede Typisierung eine, bis auf Variablenumbenennungen und

die Äquivalenz von Restriktionsmengen, eindeutige Normalform unter dem transitiven Abschluß von $\mapsto_G$ besitzt. Darüber hinaus gilt für Typaussagen t, t' mit $t \mapsto_G t'$, daß sowohl t eine verzögerte Instanz von t' als auch t' eine verzögerte Instanz von t ist. Somit kann $\mapsto_G$ zur Vereinfachung von Typaussagen verwendet werden, ohne daß die „principal type“ Eigenschaft der Typaussage verloren geht.

In dem um parametrische Überladungen erweiterten Typsystem müssen für die Definition der G-Subsumierung nur die Überladungsrestriktionen für die zu ersetzende Typvariable berücksichtigt werden.

Definition 6.16 (Überladene G-Subsumption). Ein atomarer Typ $r \in atoms(C)$ G-subsumiert $\alpha \in \mathcal{V}(C)$, falls die folgenden Bedingungen erfüllt sind:

$$(a)\ \mathcal{A}_C(\alpha) - \{\alpha\} \subseteq \mathcal{A}_C(r),$$
$$(b)\ \mathcal{B}_C(\alpha) - \{\alpha\} \subseteq \mathcal{B}_C(r),$$
$$(c)\ ops(\alpha) \subseteq Ops(r),$$

wobei

$$Ops(r) = \begin{cases} ops(r) & \text{falls } r \in \mathcal{V}, \\ \{o \mid o{\uparrow}r\} & \text{falls } r \in F_0. \end{cases}$$

Da $\leq_G$ für den überladenen Fall eine Teilrelation der Definition von Fuh und Mishra ist, lassen sich die Ergebnisse ohne weiteres übertragen. Wie von den Autoren angemerkt, läßt sich das Verfahren mit einem Aufwand der Ordnung $O(n^3)$ implementieren, wobei $n = |\mathcal{V}(C)|$.

Die zweite Transformation ersetzt externe Typvariablen. Zunächst benötigen wir eine Hilfsfunktion *pol*, welche beschreibt, in welche Richtung eine Typvariable α konvertiert wird, wenn eine Typausdruck τ in einen strukturell äquivalenten Typ τ' konvertiert wird.

Definition 6.17 (Polarität).

$$pol(\alpha, \tau) = \begin{cases} \{1\} & \text{falls } \tau = \alpha \in \mathcal{V}, \\ \emptyset & \text{falls } \tau = \beta \in \mathcal{V}, \beta \neq \alpha, \\ pol(\tau_1)^- \cup pol(\tau_2) & \text{falls } \tau = \tau_1 \to \tau_2, \\ \bigcup_{i=1}^{n} pol(\alpha, \tau_i) & \text{sonst.} \end{cases}$$

wobei $S^- = \{-x \mid x \in S\}$.

Diese Definition nimmt an, daß sämtliche Typkonstruktoren in allen Argumenten kovariant sind, nur der Funktionstypkonstruktor ist im ersten Argument kontravariant. Enthält $pol(\alpha, \tau)$ eine 1, so kommt α in τ kovariant vor und erzeugt eine Restriktion der Form $\alpha \triangleleft a$, d.h. $\tau \triangleleft \tau' \Vdash \alpha \triangleleft a$. Enthält $pol(\alpha, \tau)$ eine -1, so kommt α in τ kontravariant vor und erzeugt eine Restriktion der Form $a \triangleleft \alpha$, d.h. $\tau \triangleleft \tau' \Vdash a \triangleleft \alpha$. Man beachte, daß beide Fälle zugleich auftreten können: in $\alpha \to \alpha$ kommt α ko- und kontravariant vor. Gilt $\alpha \notin \mathcal{V}(\tau)$, liefert $pol(\alpha, \tau)$ die leere Menge.

Definition 6.18. Ein atomarer Typ $r \in atoms(C)$ S-subsumiert $\alpha \in \mathcal{V}(C)$ in τ, geschrieben $\alpha \leq_S r$ in C und τ, falls eine der folgenden Bedingungen erfüllt sind:

$$(a)\ pol(\alpha, \tau) = \{1\} \wedge \mathcal{B}_C(\alpha) - \{\alpha\} \subseteq \mathcal{B}_C(r),$$
$$(b)\ pol(\alpha, \tau) = \{-1\} \wedge \mathcal{A}_C(\alpha) - \{\alpha\} \subseteq \mathcal{A}_C(r).$$

Damit wird wiederum eine Transformationsrelation zur Vereinfachung von Typaussagen definiert: $C, \Gamma \vdash M : \tau \mapsto_S RC, R\Gamma \vdash M : R\tau$, falls $R = \{\alpha \mapsto r\}$ für $\alpha \leq_S r$ in C und $typeclosure(\Gamma, M, \tau)$, wobei

$$typeclosure(\Gamma, M, \tau) = \Gamma(x_1) \to \ldots \to \Gamma(x_n) \to \tau$$

für $dom(\Gamma) = \{x_1, \ldots, x_n\}$

Fuh und Mishra zeigen, daß jede Typisierung eine eindeutige Normalform unter $\mapsto_S$ besitzt, bis auf Variablenumbenennung und Äquivalenz von Konversionsmengen. Außerdem sind t, t' mit $t \mapsto_S t'$ wiederum äquivalente Typaussagen bzgl. der verzögerten Instanzrelation.

Die Anpassung der S-Subsumption an den überladenen Fall folgt dem gleichen Muster wie die Anpassung der G-Subsumption.

Definition 6.19 (Überladene S-Subsumption). Ein atomarer Typ $r \in atoms(C)$ S-subsumiert $\alpha \in \mathcal{V}(C)$ in τ, falls eine der folgenden Bedingungen erfüllt sind

$$(a)\ pol(\alpha, \tau) = \{1\} \wedge \mathcal{B}_C(\alpha) - \{\alpha\} \subseteq \mathcal{B}_C(r)\,,$$
$$(b)\ pol(\alpha, \tau) = \{-1\} \wedge \mathcal{A}_C(\alpha) - \{\alpha\} \subseteq \mathcal{A}_C(r)\,.$$

und zusätzlich $ops(\alpha) \subseteq Ops(r)$ gilt.

Die Polarität der in τ vorkommenden Typvariablen läßt sich in $O(|\tau|)$ Schritten berechnen und in einer Datenstruktur mit $O(1)$-Zugriff speichern. Damit wird der Aufwand zur Berechnung der S-Transformation von dem Term $O(|\mathcal{V}(C)|^3)$ dominiert.

Zum Abschluß geben wir einige im SAMPλE-System überprüfte Beispiele an, welche die Auswirkung der G- und S-Transformationen für den überladenen Fall illustrieren. Für Beispiele ohne Überladungen konsultiere man [FM89] oder [Reh97].

Die Funktionen f, g und h, mit den Definitionen

f x y ≡ **if** 3 = 4.5 **then** x **else** y

g x y ≡ **if** x = 3 **then** x **else** y

h x y ≡ **if** x < y **then** x **else** y

besitzen vor der Vereinfachung die allgemeinsten Typen

$\mathrm{f} : \alpha \to \beta \to \eta \mid \alpha \lhd \gamma,\ \beta \lhd \gamma,\ \mathrm{int} \lhd \delta,\ \mathrm{real} \lhd \delta,\ \delta\{=\}$,

$\mathrm{g} : \alpha \to \beta \to \eta \mid \alpha \lhd \gamma,\ \beta \lhd \gamma,\ \mathrm{int} \lhd \delta,\ \alpha \lhd \delta,\ \delta\{=\}$,

$\mathrm{h} : \alpha \to \beta \to \eta \mid \alpha \lhd \gamma,\ \beta \lhd \gamma,\ \alpha \lhd \delta,\ \beta \lhd \delta,\ \delta\{<\}$.

Nach der Vereinfachung ergeben sich folgende Typen:

$\mathrm{f} : \alpha \to \alpha \to \alpha$,

$\mathrm{g} : \alpha \to \alpha \to \alpha \mid \mathrm{int} \lhd \alpha,\ \alpha\{=\}$,

$\mathrm{h} : \alpha \to \beta \to \eta \mid \alpha \lhd \gamma,\ \beta \lhd \gamma,\ \alpha \lhd \delta,\ \beta \lhd \delta,\ \delta\{<\}$.

Das letzte Beispiel zeigt, daß sich manche Typisierungen nicht vereinfachen lassen.

Die folgenden Beispiele zeigen, daß durch die Hinzunahme von strukturellen Konversionen die Leistungsfähigkeit der Typsystems tatsächlich gesteigert wird.

Der Ausdruck 3+4.5 ist in SAMPλE mit dem Typ $real|\emptyset$ typisierbar (da $int \lhd real$ eine gültige Konversion und $real \to real \to real$ eine Überladungsinstanz von + ist). Dies ist weder in einem reinen parametrischen Überladungssystem noch in einem reinen Konversionssystem möglich.

Mit dem kombinierten System kann nun auch eine generische Funktion zu Addition von Listen beliebiger Zahltypen definiert werden: $add \equiv foldl\ (+)\ 0$. Die Funktion add hat den polymorphen Typ $\forall\alpha.list(\alpha) \to \alpha | int \lhd \alpha, \alpha\{+\}$, d.h., add kann auf alle Listen angewendet werden, deren Elementtyp ein Subtyp von int ist und für den

der überladene Operator + definiert ist. Man beachte, daß *add* auch auf leere Listen angewendet werden kann, im rein parametrisch überladenen Fall müßte man entweder den Listenparameter auf nicht leere Listen beschränken, oder sich auf die Summation von ganzen bzw. reellen Zahlen durch die Wahl von 0 bzw. 0.0 beschränken.

6.4 Diskussion

Die Literatur zum Thema Typinferenz mit Subtypen ist zu umfangreich, als daß eine vollständige Würdigung im Rahmen der vorliegenden Arbeit möglich wäre. Wir beschränken uns daher auf Typsysteme mit strukturellen Konversionen.[11]

Fuh und Mishra präsentierten in [FM88] einen Algorithmus, der die Erfüllbarkeit von Konversionsbedingungen durch eine Überprüfung der Restriktionsmenge auf Inkonsistenzen garantieren sollte. Wie sich später herausstellte [WO89], war der Algorithmus unvollständig, d.h. nicht alle unerfüllbaren Restriktionsmengen wurden erkannt.

In der Veröffentlichung des Autors über eingeschränkte Typen [Kae92] mußte aus Platzgründen auf den Algorithmus zur Berechnung einer genauesten Vereinfachung für atomare Restriktionsmengen verzichtet werden, obwohl die Algorithmen schon vollständig implementiert waren und sich im praktischen Einsatz des SAMPλE - Systems bewährt hatten. Daher wurde das Thema später von Henglein und Rehof für reine Konversionsprobleme ohne Überladungen in [HR97] erneut aufgegriffen. Die Autoren entwickelten genau die gleichen Algorithmen zur Berechnung der Lösbarkeit und einer genauesten Vereinfachung, wie die in Abschnitt 6.1 vorgestellten.

Die Kombination von Konversionstypisierung mit polymorphen LET-Konstrukt wurde von Henglein in [Hen96] untersucht. Henglein verwendet dazu eine modifizierte Instanzrelation für Typaussagen, die sognannte „halbstarke“ Instanz, die sich von der verzögerten Instanz von Fuh und Mishra durch die Bedingung $\Gamma' = S\Gamma'$ statt $C' \Vdash \Gamma' \triangleleft S(\Gamma)$ unterscheidet. In dieser Arbeit wurden auch Reduktionseigenschaften behandelt und gezeigt, daß Konversionstypisierungen für strukturelle Konversionen

[11] Für die Typisierung objektorientierter Sprachen mit einfacher Vererbung benötigt man die Existenz eines größten Typs, in den alle anderen, also auch nicht strukturell äquivalente, konvertierbar sind. Mithin gilt $t \triangleleft Object$ für beliebige t und damit kann der wichtige Schritt der strukturellen Anpassung von Konversionsrestriktionen nicht mehr erfolgen. In rein funktionalen Sprachen scheint ein solcher Typ nicht zu existieren; daher nennt Reynolds ihn wohl auch „Unsinns-Typ“ [Rey85].

abgeschlossen sind unter $\beta\eta$-Reduktion und let-Konversion, wobei die Kontravarianz der Konversion von Funktionstypen entscheidend für die Herleitung der Konsistenz der β-Reduktion ist.

Hoang und Mitchell [HM95] untersuchten die Komplexität von Konversiontypisierungen in einem Kalkül ohne polymorphes LET-Konstrukt unter wohldefinierten Instanzrelationen und zeigen u.a., daß es Terme gibt, deren Typisierung Restriktionsmengen erfordern, die linear mit der Größe der Terme wachsen. Bemerkenswert ist dabei, daß dieses Resultat unabhängig von der verwendeten Instanzrelation ist.

Minimale Typisierungen, d.h. Typisierungen mit einer minimalen Anzahl von Typvariablen und Restriktionen für atomare Konversionen, wurden von Rehof in [Reh97] untersucht. Der Autor wies nach, daß minimale Typisierungen bzgl. der verzögerten Instanzrelation immer existieren, daß die von Fuh und Mishra definierten Transformationen jedoch nicht immer eine minimale Typisierung liefern. Der Grund dafür ist die Existenz von Instanzen, welche die gleichzeitige Substitution mehrerer Typvariablen erfordern. Die G- und S-Transformation substituiert jedoch immer nur eine einzige Typvariable. Darüber hinaus zeigt Rehof, daß Lambda-Terme existieren, deren minimale Typisierung Typausdrücke und Konversionsgraphen mit 2^n verschiedenen Typvariablen erfordern und verbessert somit das o.a. Resultat von Hoang und Mitchell. Allerdings wird für die Herleitung dieses Ergebnisses die Basistypkonversionsordnung $\{a \lhd c, a \lhd d, b \lhd d, b \lhd c\}$ benötigt, die offensichtlich keinen Verband bildet. Somit bleibt unklar, inwieweit das Resultat auch für Konversionsordnungen gilt, deren Basistypkonversionen einen Verband bilden. Darüber hinaus muß man auch hier anmerken, daß in der Praxis solche Konversionsprobleme in der Regel nicht auftreten.

Die einzige dem Autor bekannte Arbeit, die sich mit der Kombination von Überladungen und Konversionen beschäftigt, ist [Smi94]. Smith präsentiert einen Inferenzalgorithmus zur Berechnung eines allgemeinsten Typs, der in etwa zwischen den Algorithmen $\mathcal{C}$ und $\mathcal{D}$ liegt: Vereinfachungen der berechneten Restriktionsmengen werden dort nur im LET-Konstrukt **let** $x = M_1$ **in** M_2 nach der Analyse von M_1 erlaubt, bevor M_2 analysiert wird, und nur dann, wenn der berechnete Typ und die Restriktionsmenge keine in der Umgebung gebundenen Typvariablen enthalten.

Der Vereinfachungsalgorithmus für eine Typisierung $C, \Gamma \vdash M : \tau$ ist folgendermaßen aufgebaut: zunächst werden alle Konversionsrestriktionen strukturell angepasst, um

atomare Konversionsrestriktionsmengen zu erhalten. Danach werden Zyklen eliminiert, da Smith, ebenso wie im SAMPλE-Sytem, eine partielle Konversionsordnung voraussetzt. Anschließend werden die folgenden Schritte solange ausgeführt, bis keine weitere Vereinfachung mehr möglich ist: suche eine Variable $\alpha \in (\mathcal{V}(\tau|C) - \mathcal{FV}(\Gamma))$ und einen atomaren Typausdruck b, sodaß $C \Vdash \{\alpha \mapsto b\}C \cup \{\alpha \mapsto b\}\tau \lhd \tau$ gilt, ersetze die Variable in C und τ durch b, entferne erfüllte Restriktionen und berechne die transitive Reduktion des so erhaltenen Graphen.

Smith erläutert sein Verfahren an einer Reihe von Beispielen, die vermuten lassen, daß sein Algorithmus die gleichen Vereinfachungen berechnet, wie das von uns vorgestellte Verfahren. Anders als Fuh und Mishra, gibt Smith jedoch keinen formalen Korrektheitsbeweis an, der zeigen würde, daß die transformierten Typisierungen äquivalent bzgl. der Instanzrelation auf Typisierungen sind. Darüber hinaus ist das Verfahren wesentlich ineffezienter als die Algorithmen von Fuh und Mishra: in jedem Schritt stehen $O(|\mathcal{V}(C)| \cdot |C|)$ Paare zur Auswahl, und die Überpüfung der Implikationsrelation erfordert mindestens $O(|C|^2)$ Schritte, sodaß man einen Aufwand von $O(|C|^4)$ annehmen kann. Auch dazu fehlt jede Aussage in [Smi94].

Smith erwähnt, daß bei einer Einschränkung der Überladungsinstanzdefinitionen auf die schon in Abschnitt 3.3 auf Seite 50 erwähnte Typkonstruktoreigenschaft ausreicht, um die Erfüllbarkeit von Restriktionsmengen mit Überladungen und Konversionen zu garantieren, schließt jedoch mit der Bemerkung: “But to get an efficient algorithm it will be necessary to restrict the subtype relation. This remains an area of further study”. Mit der vorliegenden Arbeit haben wir gezeigt, daß die Typkonstruktoreigenschaft keine notwendige Bedingung für die Entscheidbarkeit der Erfüllbarkeit ist und auch nicht ausreicht, um die Lösbarkeit von SAMPλE-Restriktionsmengen zu entscheiden. Darüber hinaus haben wir Bedingungen formuliert, die eine effiziente Lösbarkeit von Restriktionsproblemen mit Überladungen und strukturellen Konversionen garantieren.

Kapitel 7

Rekursive Typen

Rekursive Typen entstehen auf natürliche Weise als Lösungen von Unifikationsproblemen der Form $\alpha = \tau(\alpha)$, wobei $\tau(\alpha)$ ein Term ist, der α als echten Teilterm enthält. Dieses Unifikationsproblem besitzt einen eindeutig bestimmten allgemeinsten Unifikator $\sigma = [t/\alpha]$, mit $t = \tau(\tau(\tau(...)))$, wenn man den Termbegriff auf unendliche Terme erweitert.

Im Kontext der Typkonzepte für funktionalen Programmiersprachen sind unendliche Typen aus mindestens zwei Gründen von Interesse: Zum einen bieten sie die Möglichkeit Selbstapplikation und damit auch die in [Bar81, Fel88] definierten Fixpunktkombinatoren Υ , Θ, und Υ_{cbv} zu typisieren, zum anderen erlauben sie die Verwendung des aus der denotationalen Semantik bekannten Typkonstruktors disjunkte Summe [Sto77, Sch86].

Dem Fixpunktkombinator

$$\Upsilon \equiv \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))$$

wird in den üblichen denotationalen Modellen des Lambda-Kalküls eine Funktion zugeordnet, die eine als Parameter gegebene Funktion f auf ihren kleinsten Fixpunkt $\underline{fix}f$ abbildet. Es liegt daher nahe, Υ den endlichen Typ

$$\Upsilon : \forall\alpha.(\alpha \to \alpha) \to \alpha$$

zuzuordnen und Υ als vordefinierten Bezeichner bzw. Konstante zu betrachten. Eine andere Möglichkeit ist die Einführung zwei neuer Sprachkonstrukte

$$M ::= \ldots \mid \textbf{fix}\ x.M \mid \textbf{letrec}\ x = M_1\ \textbf{in}\ M_2$$

mit den Typisierungsregeln

$$\text{[FIX]}\quad \frac{\Gamma + [x:\tau] \vdash M : \tau}{A \vdash \textbf{fix}\ x.M : \tau}$$

und

$$[\text{LETREC}]\ \frac{\Gamma + [x:\tau_1] \vdash M_1 : \tau_1 \qquad \Gamma + [x:\tau_1] \vdash M_2 : \tau_2}{\Gamma \vdash \textbf{letrec}\ x = M_1\ \textbf{in}\ M_2 : \tau_2}$$

Sowohl die Semantik als auch die Typisierungsregeln dieser Konstrukte lassen sich direkt aus den folgenden Definitionen ableiten :

$$\begin{aligned} &\textbf{fix}\ x.E && \stackrel{\text{def}}{=} \Upsilon(\lambda x.E) \\ &\textbf{letrec}\ x = E_1\ \textbf{in}\ E_2 && \stackrel{\text{def}}{=} \textbf{let}\ x = \textbf{fix}\ x.E_1\ \textbf{in}\ E_2 \end{aligned}$$

Im Hindley-Milner Typsystem kann die rechte Seite der obigen Definition von Υ jedoch nicht typisiert werden, da man x die Lösung der Gleichung $type(x) = type(x) \to \alpha$ als Typ zuordnen müßte. Läßt man solche Lösungen zu, dann bereitet die Typisierung von Υ keine Schwierigkeiten:

Sei $t = t \to \alpha$, $A = [x : t \to \alpha]$, $B = [f : \alpha \to \alpha]$ und $E = \lambda x.f(xx)$. Dann gilt mit $A \cup B \vdash x : t$ auch $A \cup B \vdash x : t \to \alpha$ und $A \cup B \vdash xx : \alpha$. Daraus folgt $A \cup B \vdash f(xx) : \alpha$, bzw. $B \vdash E : t \to \alpha$. Wegen $t = t \to \alpha$ gilt aber auch $B \vdash E : t$ und somit erhält man nach Anwendung der [APP]- und [ABS]-Regel $B \vdash EE : \alpha$ bzw. $[] \vdash \lambda f.EE : (\alpha \to \alpha) \to \alpha$.

Interessanterweise führen rekursive Typen dazu, daß jeder geschlossene λ-Term typisierbar ist. Es gilt nämlich

Satz 7.1 ([Cop85]). *Sei Γ eine Typannahme, die jedem $x \in \mathcal{FV}(M)$ den Typ zuordnet, der die Gleichung $t = t \to t$ erfüllt, dann ist $\Gamma \vdash M : t$ eine gültige Typisierung.*

Der Beweis erfolgt durch Induktion über die Definition der Typisierbarkeit. Natürlich ist dieses Ergebnis für Sprachen mit Typkonstanten ohne Belang.

Rekursive disjunkte Summen entstehen u.A. bei der Charakterisierung der semantischen Bereiche in denotationalen Semantikdefinitionen. So ist z.B. die Gleichung

$$V = Int \oplus [V \mapsto V]$$

eine Beschreibung des semantischen Bereichs einer Sprache mit elementarem Typ ganzer Zahlen und Funktionen höherer Ordnung. Die entsprechende Typgleichung lautet:

$$type(V) = int \oplus (type(V) \to type(V))$$

7.1 Reguläre Bäume

Die Darstellung der Theorie regulärer Bäume orientiert sich im Wesentlichen an Courcelle[Cou83].

Definition 7.2 (Bäume). Ein geordnetes Alphabet ist ein Paar (F, ρ) wobei F eine nicht notwendigerweise endliche Menge ist und $\rho : \mathbb{N} \to \mathbb{N}$ jedem $f \in F$ einen Rang $\rho(f)$ zuordnet. Die Menge der Symbole mit Rang i $\{f \in F | \rho(f) = i\}$ wird mit F_i bezeichnet. Ein Baum über einem geordneten Alphabet ist eine partielle Abbildung $t : \mathbb{N}_+^* \to F$ deren Urbild $dom(t)$ den folgenden Bedingungen genügt:

1. $dom(t)$ ist präfix-abgeschlossen, d.h. $\forall \mu, \nu \in \mathbb{N}_+^*$ gilt

$$\mu\,\nu \in dom(t) \Rightarrow \mu \neq \epsilon \Rightarrow \mu \in dom(t)$$

2. $\forall \mu \in \mathbb{N}_+^*, 1 \leq i \leq j : \mu\,i \in dom(t) \Rightarrow \mu\,i \in dom(t)$

3. falls $t(\mu) = f$ und $\rho(f) = k$ dann gilt: $\mu\,i \in dom(t) \iff 1 \leq i \leq k$

Wir verwenden die folgenden Bezeichnungen:

$M^\infty(F)$	die Menge aller Bäume über F.
$M(F)$	die Menge aller endlichen Bäume über F: $M(F) = \{t \in M^\infty(F) \mid \lvert dom(t)\rvert < \infty\}$
t/μ	der Teilbaum t' von t mit Wurzelposition μ: $t' = \lambda\nu.t(\mu\nu)$
$subtrees(t)$	die Menge aller Teilbäume von t: $\{t/\mu \mid \mu \in dom(t)\}$

Definition 7.3 (Reguläre Bäume). Ein Baum t heißt regulär, falls er nur endlich viele verschiedene Teilbäume enthält. Die Menge $R(F)$ der regulären Bäume über F ist gegeben durch: $R(F) = \{t \in M^\infty(F) \mid \lvert subtrees(t)\rvert < \infty\}$.

Satz 7.4. *Die folgenden Eigenschaften regulärer Bäume sind eine unmittelbare Konsequenz der Definitionen:*

1. $M(F) \subset R(F) \subset M^\infty(F)$, *falls* F *mindestens zwei Symbole mit einem Rang* ≥ 2 *enthält.*

2. *Jeder Teilbaum eines regulären Baums ist regulär.*

3. *Die Anzahl der Symbole eines regulären Baums ist endlich.*

4. *Die Menge der regulären Bäume ist abgeschlossen unter den F-Operationen:*

$$t_1, \ldots, t_n \in R(F),\ f \in F_n \Rightarrow f(t_1, \ldots, t_n) \in R(F).$$

Von besonderem Interesse für die Erweiterung der Resultate aus Kapitel 4 ist die Tatsache, daß reguläre Bäume durch Gleichungssysteme dargestellt werden können: Ein *reguläres Gleichungssystem* ist eine endliche Menge von Gleichungen der Form

$$S = \langle x_1 = u_1, \ldots, x_n = u_n \rangle.$$

über den Unbekannten $X = \{x_1, \ldots, x_n\}$. Dabei sind die u_i Elemente aus $M(F \cup X)$, d.h. sie haben die Form f für $f \in F_0$ oder $f(x_{i_1}, \ldots, x_{i_k})$ für ein $f \in F_k$ und $x_{i_k} \in 1..n$. Die Lösung eines solchen Gleichungssystems ist ein n-Tupel $(t_1, \ldots, t_n)$ mit der Eigenschaft $t_i = u_i[t_1/x_1, \ldots, t_n/x_n]$ für alle i.

Satz 7.5. *Jedes reguläre Gleichungssystem hat eine eindeutige Lösung in $R(F)$. Jeder reguläre Baum ist Teil der Lösung eines regulären Gleichungssystems.*

Beweis. Zu Teil 1 siehe [Cou83]. Das einen regulären Baum t repräsentierende Gleichungssystem $S(t)$ kann auf folgende Weise konstruiert werden: Sei $\{s_1, \ldots, s_n\} = \mathit{subtrees}(t)$, wobei o.B.d.A $s_1 = t$. Für jedes s_i sei x_i eine Variable. Ist s_i von der Form $f(s_{i_1}, \ldots, s_{i_k})$ dann enthält $S(t)$ die Gleichung $x_i = f(x_{i_1}, \ldots, x_{i_k})$. Da $S(t)$ nach Konstruktion regulär ist, gibt es eine eindeutige Lösung $(t_1, \ldots, t_n)$ für $S(t)$ mit $t_1 = t$. □

Eine alternative Darstellung für reguläre Bäume erhält man über die sogenannten rationalen Ausdrücke.

Definition 7.6 (Rationale Ausdrücke). Sei Q eine (abzählbare) Menge von Variablen. Ein *rationaler Ausdruck* r ist ein Element der durch die abstrakte Syntax

$$r := v \mid \boldsymbol{\mu} v.r \mid f(r_1, \ldots, r_n)$$

induzierten freien Termalgebra $R_\Sigma(Q)$. Ein rationaler Ausdruck ist *wohlgeformt*, falls für jeden Teilausdruck der Form $\boldsymbol{\mu} v.r$ gilt: $v \in \mathcal{FV}(r)$ und r ist von der Form $f(r_1, \ldots, r_n)$. Hierbei sind $\mathcal{V}$ bzw. $\mathcal{FV}$ in der üblichen Weise definiert:

$$\begin{aligned} \mathcal{V}(v) &= \{v\} & \mathcal{FV}(v) &= \{v\} \\ \mathcal{V}(f(\overline{r_n})) &= \mathcal{V}(r_1) \cup \cdots \cup \mathcal{V}(r_n) & \mathcal{FV}(f(\overline{r_n})) &= \mathcal{FV}(r_1) \cup \cdots \cup \mathcal{FV}(r_n) \\ \mathcal{V}(\boldsymbol{\mu} v.r) &= \mathcal{V}(r) \cup \{v\} & \mathcal{FV}(\boldsymbol{\mu} v.r) &= \mathcal{FV}(r) - \{v\} \end{aligned}$$

Die Anwendung von Substitutionen $S(r)$ ist in der üblichen Weise definiert. Dabei muß nur sichergestellt werden, daß S keine durch $\boldsymbol{\mu}$ gebunden Variablen involviert. Dies kann durch α-Konversion der durch $\boldsymbol{\mu}$ gebundenen Variablen immer erreicht werden. Wir identifizieren daher reguläre Ausdrücke, die sich nur in den Namen der gebunden Variablen unterscheiden.

Ein wohlgeformter regulärer Ausdruck steht für einen regulären Baum, den man durch unendliche Expansion der $\boldsymbol{\mu}$-Teilausdrücke erhält. Dazu definiert man die Expansion $\mathcal{T}(r)$ von r wie folgt:

$$\begin{aligned} \mathcal{T}(v) &= v \\ \mathcal{T}(f(\overline{t_n})) &= f(\mathcal{T}(t_1), \ldots, \mathcal{T}(t_n)) \\ \mathcal{T}(\boldsymbol{\mu} v.t) &= \mathcal{T}(t[\boldsymbol{\mu} v.f(\overline{t_n})/v]) \end{aligned}$$

Nach Definition einer Äquivalenzrelation $\approx$ mit $t_1 \approx t_2 \iff \mathcal{T}(t_1) = \mathcal{T}(t_2)$ kann man Typdeduktionssysteme durch Erweiterung mit einer Regel der Form

$$[\text{REC}] \quad \frac{\Gamma \vdash M : \tau \qquad \tau \approx \tau'}{\Gamma \vdash M : \tau'}$$

direkt auf regulären Ausdrücken definieren (siehe z.B. [Bar92]).

Jeder rationale Ausdruck kann auf einfache Weise in ein reguläres Gleichungssystem transformiert werden und umgekehrt. Wir geben dazu zwei Funktionen *mksys* bzw. *mkexpr* an, die das Gewünschte leisten:

$$mksys(r) = \begin{cases} (v, \emptyset) & \text{falls } t = v \in Q, \\ mkeq(v, r') & \text{falls } r = \boldsymbol{\mu} v.r', \\ mkeq(v', r) & \text{für eine neue Variable } v' \text{ sonst,} \end{cases}$$

wobei

$$\begin{aligned} mkeq(v, f(\overline{r_n})) &= (v, S \cup \{v = f(\overline{r'_n})\}) \\ &\quad \text{mit } S = S_1 \cup \cdots \cup S_n \\ &\quad \text{und } (r'_i, S_i) = mksys(r_i) \text{ für } i = 1..n. \end{aligned}$$

Für die Umkehrung benötigen wir zunächst eine Hilfsdefinition. Sei E ein reguläres Gleichungssystem. Eine Variable x kommt als Teilterm einer Variablen y in E vor ($x \ll_E y$), falls eine Gleichung $y = f(..x..)$ in E existiert. Die transitive Hülle der Relation $\ll_E$ bezeichnen wir mit $\ll_E^+$. Der Index E kann weggelassen werden wenn aus dem Kontext eindeutig hervorgeht welches Gleichungssystem gemeint ist. Die Funktion *mkexpr* wird damit wie folgt definiert:

$$mkexpr(v, E) = me(v, \emptyset)$$

wobei

$$me(v, Z) = \begin{cases} v & \text{falls } v \in Z \vee v \notin \mathcal{V}(E), \\ \boldsymbol{\mu} v.f(\overline{r_n}) & \text{falls } v = f(\overline{x_n}) \in E,\ v \ll_E^+ v \\ & \text{und } r_i = me(x_i, Z \cup \{v\}), \\ f(\overline{r_n}) & \text{falls } v = f(\overline{x_n}) \in E,\ \neg(v \ll_E^+ v) \\ & \text{und } r_i = me(x_i, Z \cup \{v\}). \end{cases}$$

Bei der Umwandlung eines Gleichungssystems in einen rationalen Ausdruck und wieder zurück können redundante Gleichungen entstehen. Betrachtet man den Term t, der als erste Komponente der Lösung des Gleichungssystems

$$S = \langle x = y \times y,\ y = y \to z,\ z = int \rangle$$

auftritt, so wird t durch den rationalen Ausdruck

$$(\boldsymbol{\mu} y.y \to int) \ \times \ (\boldsymbol{\mu} y.y \to int)$$

äquivalent repräsentiert. Die Rückumwandlung dieses Ausdrucks resultiert jedoch in dem Gleichungssystem

$$S' = \langle x = y_1 \times y_2,\ y_1 = y_2 \to z_1,\ y_2 = y_2 \to z_2,\ z_1 = int,\ z_2 = int \rangle$$

das gegenüber dem Ausgangssystem S zwei zusätzliche Variablen (und damit auch Gleichungen) enthält. Man sieht, daß rationale Terme keine minimale Darstellungsform für reguläre Bäume sind. Für das Beispiel ist klar, wie man S' in S überführen kann: nämlich durch Ersetzen von y_2, z_2 durch y_1 bzw. z_1 und Eliminieren der dann zweifach vorkommenden Gleichungen $y_1 = y_1 \to z_1$ bzw. $z_1 = int$. Allgemein erhält man den folgenden

Satz 7.7. *Sei $\sim$ eine Äquivalenzrelation auf den in E vorkommenden Variablen mit der Eigenschaft*

$$\begin{aligned} x \sim y \wedge x = f(\overline{x_n}) \in E \wedge y = g(\overline{y_n}) \in E \\ \Rightarrow g = f \wedge \forall i = 1..n : x_i \sim y_i \end{aligned} \tag{7.1}$$

dann ist die Reduktion von E nach $E/_\sim$ ein zu E äquivalentes Gleichungssystem, d.h., zu jeder Variablen $x \in E$ gibt es eine Variable y in $E/_\sim$ sodaß $x \sim y$ und $t_x = t_y$, wobei t_x die x entsprechende Komponente der Lösung des Gleichungssystems E und t_y die y entsprechende Komponente der Lösung des Gleichungssystems $E/_\sim$ ist.

Beweis. Siehe [Cou83]. □

Die Minimierung des Gleichungssystems erfolgt durch Berechnung der gröbsten Äquivalenzrelation, die Gleichung (7.1) erfüllt. Dies kann mit dem Aufwand $O(n \cdot \log n)$ implementiert werden (siehe z.B. [AHU74, Kapitel 4.13]).

7.2 Restriktionsauflösung

Wir beginnen mit den positiven Resultaten: da $R(F)$ einen vollständigen metrischen Raum bildet, können für endliche Bäume definierte Prädikate eindeutig auf unendliche, und damit auch auf reguläre Bäume fortgesetzt werden. Für parametrische Überladungen haben wir schon in Abschnitt 3.4.3 einen unitären Unifikationsalgorithmus für endliche Terme angegeben. Daher ist für diesen Fall die Existenz eines allgemeinsten Typs mit atomar gelöster Restriktionsmenge gesichert. Darüber hinaus ist auch die denotationale Semantik regulärer Typausdrücke eindeutig definiert, wie die Arbeit von Mc. Queen et. al. [MPS84] gezeigt hat.

Lemma 7.8. *Seien $t_1, \ldots, t_n$ unendliche, strukturell ähnliche Bäume und p eine zerlegbares Prädikat. Die Restriktion $p(\overline{t_n})$ ist äquivalent zu einer unendlichen Menge atomarer Restriktionen $\bigwedge_{i \in I} p_i(a_{i_1}, \ldots a_{i_n})$, wobei die a_i als Blätter der Bäume $t_1, \ldots, t_n$ vorkommen. Falls die t_i regulär sind, ist I endlich.*

Beweis. Der erste Teil ist offensichtlich. Die Endlichkeit von I für reguläre Bäume folgt aus der Tatsache, daß jeder reguläre Baum nur endlich viele verschieden Teilbäume hat. □

Zur Berechnung der endlichen Menge atomarer Restriktionen geben wir zwei Transformationsregeln an. Sie unterscheiden sich von den in Abschnitt 5.3 angegebenen durch die Hinzunahme einer Restriktionsmenge H, welche die schon betrachteten Restriktionen aufnimmt. Damit wird die Termination der Transformation sichergestellt.

$$
\begin{array}{ll}
H, C \cup \{c\} & \Longrightarrow H, C \\
 & \text{falls } c \in H \text{ und } \exists \tau \in \mathit{args}(c) : \tau = \boldsymbol{\mu}\alpha.\tau' \\
H, C \cup \{p(f_1(\overline{\tau_n}_1), \ldots, f_m(\overline{\tau_n}_m))\} & \Longrightarrow H \cup \{p(f_1(\overline{\tau_n}_1), \ldots, f_m(\overline{\tau_n}_m))\}, C \cup S(\overline{r_k}) \\
 & \text{falls } p(f_1(\overline{\alpha_n}_1), \ldots, f_m(\overline{\alpha_n}_m)) \longrightarrow\!\!\!\!\!\rightarrow \overline{r_k} \in R \\
 & \text{und } S = \{\alpha_{11} \mapsto \tau_{11}, \ldots, \alpha_{nm} \mapsto \tau_{nm}\} \\
 & \text{und } \{p(f_1(\overline{\tau_n}_1), \ldots, f_m(\overline{\tau_n}_m))\} \notin H
\end{array}
$$

Proposition 7.9. *Für Restriktionsmengen über strukturell ähnlichen regulären Bäumen, ist $\Longrightarrow$ eine terminierende Transformationsrelation. Falls $\emptyset, C \Longrightarrow H, C'$ und H, C' eine Normalform unter $\Longrightarrow^*$ ist, dann ist C' eine atomare Restriktionsmenge äquivalent zu C oder C ist nicht lösbar.*

Betrachtet man nur strukturelle Äquivalenz, wird die zweite Regel wie folgt abgeändert:

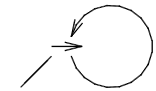
$$
\begin{array}{ll}
H, C \cup \{p(f(\overline{\tau}_1), \ldots, f(\overline{\tau}_n))\} & \Longrightarrow H \cup \{p(f(\overline{\tau_n}_1), \ldots, f(\overline{\tau_n}_m))\}, C \cup S(\overline{r_k}) \\
 & \text{falls } p(f(\overline{\alpha_n}_1), \ldots, f(\overline{\alpha_n}_m)) \longrightarrow\!\!\!\!\!\rightarrow \overline{r_k} \in R \\
 & \text{und } S = \{\alpha_{11} \mapsto \tau_{11}, \ldots, \alpha_{nm} \mapsto \tau_{nm}\} \\
 & \text{und } \{p(f(\overline{\tau_n}_1), \ldots, f(\overline{\tau_n}_m))\} \notin H
\end{array}
$$

Der essentielle Schritt bei der Transformation eines beliebigen Restriktionsproblems in ein äquivalentes, atomar gelöstes, ist die Ersetzung von Typvariablen durch strukturell äquivalente bzw. ähnliche Muster. Wie wir gezeigt haben, ist dieser Schritt für endliche Terme vollständig: ist τ ein strukturelles Muster von τ', also $\tau \overset{\text{se}}{\sim} \tau'$ dann ist jede Instanz von τ' auch eine Instanz von τ. Bedauerlicherweise gilt dies nicht für reguläre Bäume.

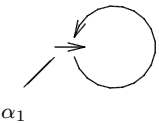
Definition 7.10. Zwei unendliche Bäume t, t' sind strukturell ähnlich genau dann, wenn $dom(t) = dom(t)$. Sie sind strukturell äquivalent, wenn zusätzlich noch $t(\pi) = t'(\pi)$ für alle inneren Knoten π gilt, d.h. für all $\pi \in dom(t) : \exists \nu \neq \epsilon : \pi \cdot \nu \in dom(t)$.

Man sieht leicht, daß diese Definition eine konservative Erweiterung der entsprechenden endlichen Fälle darstellt.

Man betrachte nun das Restriktionsproblem $\alpha \triangleleft t$, wobei t die Lösung des Gleichungssystems $\beta = int \to \beta$ ist.[1] Graphisch dargestellt:

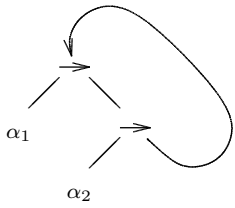


Um dieses Problem in ein äquivalentes, atomar gelöstes umzuwandeln, müßte α durch eine Darstellung aller Bäume ersetzt werden, die strukturell äquivalent zu t sind. Als ersten Ansatz könnte man es mit t_1 versuchen, der Lösung des Gleichungssystems $\alpha = \alpha_1 \to \alpha$.[2]

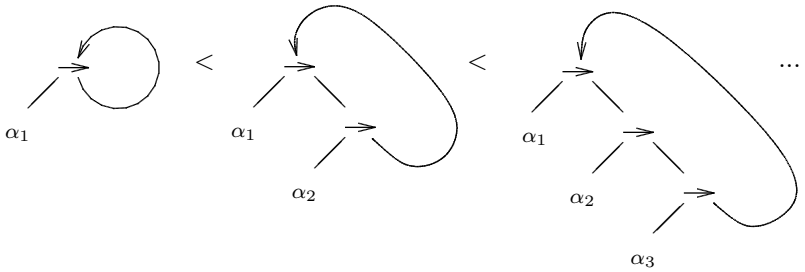


Klarerweise ist t_1 strukturell äquivalent zu t. Aber das gilt auch für t_2, der Lösung von $\alpha = \alpha_1 \to (\alpha_2 \to \alpha)$.[3]

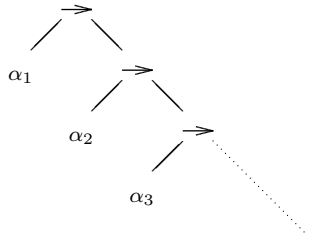
[1] $t = \boldsymbol{\mu}\beta.int \to \beta.$
[2] $t_1 = \boldsymbol{\mu}\beta.\alpha_1 \to \beta.$
[3] $t_2 = \boldsymbol{\mu}\beta.\alpha_1 \to (\alpha_2 \to \beta).$



Nun ist t_1 eine echte Instanz von t_2, d.h. $t_1 \leq t_2 \wedge t_2 \not\leq t_1$, und somit enthält die Menge der zu t strukturell äquivalenten Bäume die folgende, strikt ansteigende Sequenz regulärer Bäume:



die zu dem folgenden unendlichen, aber nicht regulären Baum t_∞ konvergiert:



Mit anderen Worten: zu jeder endlichen Menge strukturell äquivalenter regulärer Bäume $\{t_1, \ldots, t_n\}$, kann man einen regulären Baum t' konstruieren, mit $dom(t') = dom(t_i) \wedge t_i \leq t' \wedge t' \not\leq t_i$ für alle $i = 1..n$. Insgesamt folgt daraus:

Lemma 7.11. *Sei t eine regulärer Baum. Dann gibt es keine Repräsentation der*

zu t strukturell äquivalenten regulären Bäume durch ein endliche Menge regulärer Muster.[4]

Damit bricht der Ansatz der Reduktion beliebiger Restriktionsprobleme in endliche atomare Restriktionsmengen durch Berechnung einer strukturelle Äquivalenz erzwingenden Substitution in sich zusammen.

Vermutung 7.12. *Für Restriktionstypsysteme über zerlegbaren, induktiv definierten Prädikaten kann die Existenz eines allgemeinsten Typs mit atomarer Restriktionsmenge nicht garantiert werden.*

Verzichtet man auf Vollständigkeit des Typinferenzalgorithmus, kann man eine Approximation von t_∞ wählen, also beispielsweise t_1, α durch t_1 ersetzen und dann mit Hilfe der Transformationsrelation $\Longrightarrow$, da der Funktionstypkonstruktor Kontravariant ist, das folgende atomar gelöste System erhalten: $int \lhd \alpha_1$. Wählt man t_2, erhält man $int \lhd \alpha_1, int \lhd \alpha_2$.

Wir geben in Abbildung 7.1 ein Regelsystem an, das diesen Transformationsprozeß für beliebige Restriktionsprobleme formal beschreibt. Dabei benutzen wir $\boldsymbol{\mu} S.\tau$ als Abkürzung für $\boldsymbol{\mu}\alpha_1 \ldots \boldsymbol{\mu}\alpha_n.\tau$, wobei $S = \{\overline{\alpha_n}\}$ und $\boldsymbol{\mu}\{\}.\tau = \tau$. Man beachte, daß $\boldsymbol{\mu}\alpha.\tau = \alpha$, falls $\alpha \notin \mathcal{V}(\tau)$.

Wir betrachten das Regelsystem an einem Beispiel: das Restriktionsproblem $\alpha \lhd f(\beta,\gamma), \beta \lhd \alpha, \delta \lhd \alpha$ hat keine endliche Lösung. Wir nehmen an, daß die Variablen α, β und δ in einer gemeinsamen Äquivalenzklasse liegen. Eine Anwendung der Ersetzungsregel S_2 führt dann zur Ersetzung von α, β und δ durch eine neue Kopie eines zu dem Ausdruck $\boldsymbol{\mu}[\alpha]_E.f(\beta,\gamma)$ strukturell äquivalenten Musters. Wegen

$$\boldsymbol{\mu}[\alpha]_E.f(\beta,\gamma) \approx \boldsymbol{\mu}\alpha.\boldsymbol{\mu}\beta.\boldsymbol{\mu}\delta.f(\beta,\gamma) \approx \boldsymbol{\mu}\beta.f(\beta,\gamma)$$

ist

$$\{\alpha \mapsto \boldsymbol{\mu}\beta.f(\beta,\gamma), \gamma \mapsto \boldsymbol{\mu}\beta.f(\beta,\gamma_2), \gamma \mapsto \boldsymbol{\mu}\beta.f(\beta,\gamma_2)\}$$

eine mögliche strukturelle Gleichheit erzwingende Substitution.

[4] Man könnte vermuten, daß bei einer endlichen Basistypmenge nur reguläre Instanzen von t_∞ konstruierbar sind; es genügen jedoch 2 Basistypen a, b, um eine monomorphe Instanz zu erzeugen, die nicht regulär ist: wähle z.B. $a, b, a, a, b, b, a, a, a, b, b, b, \ldots$ für $\alpha_1, \alpha_2, \alpha_3, \ldots$.

H	$H, E, C \cup \{c\}$	$\Longrightarrow_r H, E, C$ falls $c \in H$ und $\exists \tau \in \mathit{args}(c) : \tau = \boldsymbol{\mu}\alpha.\tau'$
L	$H, E, C \cup \{\tau = \tau\}$	$\Longrightarrow_r H, E, C$
Z	$H, E, C \cup \{f(\overline{\tau_n}) = f(\overline{\tau'_n})\}$	$\Longrightarrow_r H \cup \{f(\overline{\tau_n}) = f(\overline{\tau'_n})\}, E,$ $C \cup \{\tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n\}$ falls $\{f(\overline{\tau_n}){=}f(\overline{\tau'_n})\} \notin H$
T	$H, E, C \cup \{p(f_1(\overline{\tau_n}_1),$ $\ldots, f_k(\overline{\tau_n}_k))\}$	$\Longrightarrow_r H \cup \{p(f_1(\overline{\tau_n}_1), \ldots, f_k(\overline{\tau_n}_k))\}, E, C \cup S(\overline{r_m})$ falls $p(f_1(\overline{\alpha_n}_1), \ldots, f_k(\overline{\alpha_n}_k)) \longrightarrow\!\!\!\!\!\rightarrow \overline{r_m} \in R$ und $S = \{\alpha_{11} \mapsto \tau_{11}, \ldots, \alpha_{nk} \mapsto \tau_{nk}\}$ und $\{p(f_1(\overline{\tau_n}_1), \ldots, f_k(\overline{\tau_n}_k))\} \notin H$
S_1	$H, E, C \cup \{\alpha = \beta\}$	$\Longrightarrow_r SH, \{\beta/\alpha\}(E + \{(\alpha, \beta)\}), SC \cup \{\alpha = \beta\}$ falls $\alpha, \beta \in \mathcal{V}(C),\ \alpha \neq \beta$ und $S = \{\alpha \mapsto \beta\}$
S_2	$H, E, C \cup \{\alpha = \tau\}$	$\Longrightarrow_r S'(H), E', S \cup S'(C) \cup \{\alpha = \tau\}$ falls $\alpha \in \mathcal{V}(C),\ \tau \notin \mathcal{V}$ und $\vec{S} \in$ $\mathcal{SS}(\boldsymbol{\mu}[\alpha]_E.\tau, [\alpha]_E - \{\alpha\}, \mathcal{V}(C \cup \{\alpha = \tau\}))$ wobei $S' = \vec{S} \circ \{\alpha \mapsto \tau\}$ und $E' = E_\alpha + EQ(\{\boldsymbol{\mu}[\alpha]_E.\tau = \vec{S}(\beta) \mid \beta \in [\alpha]_E{-}\alpha\})$
A	$E, C \cup \{p(\overline{\tau_n})\}$	$\Longrightarrow_r \vec{S}H, E', S \cup \vec{S}(C \cup \{p(\overline{\tau_n})\})$ falls $\alpha, \tau \in \mathit{args}(p(\overline{\tau_n})),\ \mathit{root}(\tau) \in F_+$ und $\vec{S} \in \mathcal{SS}(\boldsymbol{\mu}[\alpha].\tau, [\alpha]_E, \mathcal{V}(C \cup \{p(\overline{\tau_n})\}))$ wobei $E' = E_\alpha + EQ(\{\boldsymbol{\mu}[\alpha]_E.\tau{=}\vec{S}(\beta) \mid \beta \in [\alpha]_E\})$
F_1	$E, C \cup \{f(\overline{\tau_n}) = g(\overline{\tau'_n})\}$	$\Longrightarrow_r E, \top$
F_2	$E, C \cup \{p(f_1(\overline{\tau_n}_1),$ $\ldots, f_k(\overline{\tau_n}_k))\}$	$\Longrightarrow_r E, \top$ falls $\nexists\ p(f_1(\overline{\alpha_n}_1), \ldots, f_k(\overline{\alpha_n}_k)) \longrightarrow\!\!\!\!\!\rightarrow \overline{r_m} \in R$

Abbildung 7.1: Terminierende Transformation in atomare Restriktionsmengen für rekursive Typen

7.3 Diskussion

Obwohl das o.a. Verfahren nur eine Annäherung an einen vollständigen Algorithmus darstellt, wurden im mehrjährigen Einsatz in der SAMPλE-Programmierumgebung keine Probleme festgestellt. Dies liegt wahrscheinlich daran, daß die meisten Programme, die „echte“ Konversionen verlangen, ohne rekusive Typen typisiert werden können und die meisten Programme die rekursive Typen erfordern, ohne Konversionen typisiert werden können. Für diese beiden Fälle ist unser Algorithmus vollständig. Und für die verbleibenden Programme, die beide Konzepte gleichzeitig benötigen, scheint die implementierte Heuristik gut zu funktionieren.

In [AC91] wurde die Frage der Entscheidbarkeit von $t \triangleleft t'$ untersucht, für monomorphe reguläre Typen t, t', d.h. $\mathcal{V}(t) = \mathcal{V}(t') = \emptyset$. Wie die vorliegende Arbeit und die u.a. Artikel zeigen, ist die Frage $\exists S : S\tau_1 \triangleleft S\tau_2$ wesentlich komplexer. Außerdem unterscheidet sich die betrachtete Konversionstheorie von den strukturellen Konversionen durch die Existenz eines minimalen und eines maximalen Typs. Für strukturelle Konversionen ist die Frage der Konvertierbarkeit monomorpher Typausdrücke durch unser Verfahren ebenfalls positiv beantwortet.

In [TW93] wurde gezeigt, daß die Lösbarkeit von Konversionsrestriktionsmengen für den Fall der strukturellen Konversionen für reguläre Typen in DEXPTIME liegt und damit entscheidbar ist. Der Beweis erfolgt über eine Reduktion auf das Leerheitsproblem von Büchi-Atomaten. Aufgrund von Lemma 7.1 können die Restriktionsmengen jedoch nicht auf atomar gelöste zurückgeführt werden. Damit entfällt auch die in Abschnitt 6.3 beschriebene Vereinfachung von Restriktionsmengen. Da die Größe der Restriktionsmengen linear mit der der Programmgröße wachsen kann, muß diese Methode für den praktischen Einsatz skeptisch beurteilt werden. Darüberhinaus scheint das Verfahren nicht auf Typsysteme mit Überladungen übertragbar zu sein.

Kapitel 8

Resultate und Ausblick

Ziel dieser Arbeit war die Entwicklung eines Typsystems, das Überladungen, implizite strukturelle Konversionen und reguläre Typen kombiniert. Daß dies gelungen ist, zeigen sowohl die theoretischen Ergebnisse als auch die Implementierung der Typinferenzkomponente in der SAMPλE-Programmierumgebung, die mehrere Jahre erfolgreich in Forschung und Lehre an der TU Darmstadt eingesetzt wurde.

Durch die Einschränkung auf parametrische Überladungen konnte ein polynomial zeitbeschränkter Typinferenzalgorithmus für let-freie Lambda-Terme mit überladenen Operatoren entwickelt werden, der in der praktischen Anwendung gegenüber dem parametrisch polymorphen Fall keine Nachteile erkennen läßt. Wir haben gezeigt, daß sich sämtliche Resultate des rein parametrischen Polymorphismus auf den parametrisch überladenen Fall übertragen. Außerdem erlaubt die Methode die vollständige Erweiterung auf reguläre Typen.

Da jedoch einige sinnvolle Überladungen nicht parametrisch und auch Konversionen in dem System nicht handhabbar sind, haben wir den Begriff der eingeschränkten Typen eingeführt und gezeigt, daß sich damit u.a. parametrische Überladungen, beliebige endliche Überladungen und strukturelle Konversionen darstellen lassen. Dabei ist das System der eingeschränkten Typen völlig allgemein gehalten: es erlaubt beliebige Restriktionen und hat, wie andere Autoren gezeigt haben, Anwendungen auch außerhalb der Typinferenz für Überladungen und Konversionen.

Das wichtigste Resultat der Einführung eingeschränkter Typen ist sicher die Tatsache, daß wir zeigen konnten, daß Let-Polymorphismus völlig unabhängig von der Art der Restriktionen funktioniert. Dies wurde durch die Einführung eines generischen Typdeduktionssystems mit einem zugehörigen wohldefinierten und vollständigen Inferenzalgorithmus erreicht. Damit kann man sich bei der Untersuchung neuer Restriktionstypen auf den Mechanismus der Restriktionsauflösung konzentrieren, ohne

jedesmal einen neuen Inferenzalgorithmus angeben und als korrekt und vollständig beweisen zu müssen.

Mit der Einführung des abstrakten Konzepts der Vereinfachungen gelang es, einen allgemeinen Rahmen zu schaffen, der es erlaubt, den Inferenzalgorithmus durch Anwenden vereinfachender Substitutionen parallel zum Aufsammeln der Restriktionen zu verbessern, was in einer effizienter ablaufenden Typinferenz resultiert. Darüber hinaus bietet das Konzept der allgemeinsten Vereinfachungen die Möglichkeit der Qualitätsbeurteilung des Resultats der Restriktionsauflösung: berechnet der Inferenzalgorithmus eine Restriktionsmenge, die nicht weiter vereinfacht werden kann, so ist er sicher als optimal zu bezeichnen.

Mit unseren Untersuchungen zur Lösung von Konversionsproblemen in Kombination mit parametrischen Überladungen haben wir gezeigt, daß man sehr wohl effiziente Algorithmen zur Restriktionsauflösung angeben kann, wenn man nur geeignete Einschränkungen an das Zusammenspiel von Überladungen und Konversionen stellt. Das SAMPλE-Typsystem erfüllt diese Einschränkungen. Allerdings sollte man nicht vergessen, das viele der Komplexitätsresultate in der Praxis scheinbar wenig Aussagekraft bzgl. der gespürten Effizienz der Typinferenz haben. Daher wäre es sicher sinnvoll, mit einer Implementierung zu experimentieren, die komplexere Restriktionssysteme behandeln kann.

Für das System der parametrischen Überladungen haben wir bewiesen, daß Überladungen prinzipiell sowohl zur Übersetzungszeit als auch zur Laufzeit aufgelöst werden können, beide Methoden haben Vor- und Nachteile. Die Auswahl wird dabei stark durch die Semantik der Sprache und die Implementierung des Laufzeitsystems beeinflußt. Überladungsauflösung zur Laufzeit ist nur möglich, wenn man auf kontextabhängige Überladungen (z.B. überladene Konstanten) verzichtet und wenn die Repräsentation der Werte jedes Typs zur Laufzeit ein eindeutiges Typkennzeichen besitzt. Dies impliziert insbesondere, daß isomorphe, aber unterschiedliche abstrakte Datentypen zur Laufzeit mit einer im Prinzip überflüssigen Typkennzeichnung markiert werden müssen.

Für das SAMPλE-System haben wir uns für die Auflösung von Überladungen und Konversionen zur Laufzeit entschieden. Dies ist möglich, da nur Operationen auf elementaren Datentypen eine echte Konversion erfordern. Dadurch kann es pas-

sieren, daß z.B. eine Liste sowohl Werte vom Type *int* als auch Werte vom Typ *real* enthält. Dies entspricht in etwa dem Vorgehen in gängigen LISP- und Scheme-Implementierungen. Im Unterschied zu diesen Systemen vermuten wir jedoch, daß das SAMPλE-Typsystem das Nichtauftreten von Typfehlern zur Laufzeit garantiert.

Diese Bemerkung führt uns direkt zu den offenen Problemen, bzw. Gebieten, die man noch genauer untersuchen könnte:

Ein formaler Beweis des berühmten Milner Satzes "well-typed programs can't go wrong" für das SAMPλE-System mit Konversionen und Überladungen steht noch aus. Dazu müßte man entweder ein denotationales Modell für Typausdrücke mit Restriktionsmengen entwickeln, oder wenigstens eine operationale Semantik angeben, und die Abwesenheit von Laufzeittypfehlern für typisierbare Programme beweisen. Unklar ist dabei jedoch, wie man die Abwesenheit unauflösbarer Überladungen behandelt.

Eine weitere Alternative wäre die Angabe einer Übersetzung in eine rein parametrisch polymorphe Sprache, wie wir sie für das parametrisch überladene System angegeben haben. Hier müßte gezeigt werden, daß das übersetzte Programm korrekt typisierbar ist.

Außerdem sollte man die Konsistenz von $\boldsymbol{\beta\eta}$-Reduktion und Typsystem formal untersuchen. Ausgangspunkt könnte hier die Arbeit von Henglein [Hen96] sein. Hier erwarten wir keine größeren Probleme.

Zum Abschluß die enervierendste Frage: ist die Instanzrelation für strukturelle Konversionen und parametrische Überladungen entscheidbar? Formal: gegeben 2 beliebige eingeschränkte Typen $\tau_1|C_1$ und $\tau_1|C_2$, existiert eine Substitution S, sodaß $\tau_2 = S\tau_1$ und $C_2 \Vdash SC_1$ gilt? Die Beantwortung diese Frage ist wichtig, da man dem Programmierer die Spezifikation der Typen deklarierter Bezeichner erlauben möchte. Man beachte, daß dieses Problem die Frage der Erfüllbarkeit von Restriktionsmengen beinhaltet (wähle $\tau_1 = \tau_2$ und $C_2 = \emptyset$).

Für den allgemeinen Fall beliebiger struktureller Gleichheit erzwingender Prädikate kombiniert mit parametrischen und endlichen Überladungen gibt es sicher keinen effizienten Algorithmus, da ja schon die Lösbarkeit parametrischer Überladungsrestriktion in DEXPTIME liegt. Fraglich ist, ob die in Kapitel 6 formulierten Einschränkungen an Konversionsrelation und Überladungen ausreichen, um eine (effiziente)

Lösbarkeit sicherzustellen. Die Frage ist sicher entscheidbar, wenn man nur solche Instanzierungen zuläßt, die durch atomare Typersetzungen aus dem allgemeinsten Typ entstehen.

Literaturverzeichnis

[AC91] Roberto M. Amadio und Luca Cardelli: *Subtyping Recursive Types.* In: *POPL* [POP91], Seiten 104–118. (Zitiert auf Seite 209.)

[AHU74] Alfred V. Aho, John E. Hopcroft und Jeffrey D. Ullman: *Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974. (Zitiert auf den Seiten 14, 37, 71, 177 und 203.)

[als83] alsys, 29, Avenue de Versailles, 78170 La Celle-Saint-Cloud, France: *Reference Manual for the Ada programming language*, Januar 1983. ANSI/MIL-STD 1815 A. (Zitiert auf den Seiten 35 und 151.)

[App89] Andrew W. Appel: *Runtime Tags aren't necessary.* Lisp and Symbolic Computation, 2(2):153–162, Juni 1989. (Zitiert auf Seite 94.)

[Aug93] Lennart Augustsson: *Implementing Haskell Overloading.* In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Seiten 65–73, New York, USA, Juni 1993. ACM Press. (Zitiert auf Seite 104.)

[Bak82] T. P. Baker: *A One-Pass Algorithm for Overload Resolution in Ada.* ACM Transactions on Programming Languages and Systems, 4(4):601–614, Oktober 1982. (Zitiert auf Seite 36.)

[Bar81] Hendrik Pieter Barendregt: *The Lambda Calculus: Its Syntax and Semantics*, Band 103 der Reihe *Studies in Logic and The Foundation of Mathematics.* North-Holland, 1981. (Zitiert auf den Seiten 24 und 197.)

[Bar92] H. P. Barendregt: *Lambda calculi with types.* In: D. M. Gabbai, Samson Abramski und T. S. E. Maiboum (Herausgeber): *Handbook of Logic in Computer Science.* Oxford University Press, Oxford, 1992. (Zitiert auf den Seiten 17, 25, 130 und 201.)

[BJ78] Dines Bjørner und Cliff B. Jones (Herausgeber): *The Vienna Development Method: The Metalanguage*, Band 61 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, 1978. (Zitiert auf Seite 11.)

[BJ82] Dines Bjørner und Cliff B. Jones (Herausgeber): *Formal Specification and Software Development*. Prentice-Hall, 1982. (Zitiert auf Seite 11.)

[BM72] R. S. Boyer und J. S. Moore: *The Sharing of Structure in Theorem-Proving Programs*. In: B. Meltzer und Donald Michie (Herausgeber): *Machine Intelligence*, Seiten 101–116. Edinburgh University Press, 1972. (Zitiert auf Seite 67.)

[BMS80] R. M. Burstall, D. B. MacQueen und D. T. Sannella: *HOPE: An Experimental Applicative Language*. In: *Conference Record of the 1980 LISP Conference*, Seiten 136–143. ACM, August 1980. (Zitiert auf den Seiten 1, 36 und 38.)

[BS86] Rolf Bahlke und Gregor Snelting: *The PSG System: From Formal Language Definitions to Interactive Programming Environments*. ACM Transactions on Programming Languages and Systems, 8(4):547–576, Oktober 1986. (Zitiert auf Seite 3.)

[Car87] Luca Cardelli: *Basic Polymorphic Typechecking*. Science of Computer Programming, 8(2):147–172, April 1987. (Zitiert auf Seite 28.)

[CB83] J. Corbin und Bidoit: *A Rehabilitation of Robinson's Unification Algorithm*. Information Processing, 25:909–914, 1983. (Zitiert auf den Seiten 14 und 67.)

[CDDK86] D. Clément, J. Despeyroux, T. Despeyroux und G. Kahn: *A Simple Applicative Language: Mini-ML*. In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, Seiten 13–27. ACM, 1986. (Zitiert auf den Seiten 7, 23 und 73.)

[CDO93] Gordon Cormack, Dominic Duggan und John Ophel: *Decidable Typen Reconstruction with Recursive Overloading*. draft, 1993. (Zitiert auf Seite 105.)

[Cop85] M. Coppo: *A Completeness Theorem for Recursively Defined Types.* In: Wilfried Brauer (Herausgeber): *Proceedings of the 12th Colloquium on Automata, Languages and Programming*, Band 194 der Reihe *LNCS*, Seiten 120–129, Nafplion, Greece, Juli 1985. Springer. (Zitiert auf Seite 198.)

[Cou83] Bruno Courcelle: *Fundamental Properties of Infinite Trees.* Theoretical Computer Science, 25:95–169, 1983. (Zitiert auf den Seiten 199, 200 und 203.)

[DCO96] Dominic Duggan, Gordon V. Cormack und John Ophel: *Kinded Type Inference for Parametric Overloading.* Acta Informatica, 33(1):21–68, feb 1996. (Zitiert auf Seite 106.)

[DM82] Luis Damas und Robin Milner: *Principal Type Schemes for Functional Programs.* In: *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, Seiten 207–212, Januar 1982. (Zitiert auf den Seiten 11 und 17.)

[DO94] Dominic Duggan und John Ophel: *Kinded Parametric Overloading.* Technischer Bericht CS-94-36, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada, 1994. (Zitiert auf Seite 106.)

[ES92] Margaret A. Ellis und Bjarne Stroustrup: *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, 1992. (Zitiert auf Seite 35.)

[Fel88] Matthias Felleisen: *The Theory and Practice of First-Class Prompts.* In: *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, Januar 1988. (Zitiert auf Seite 197.)

[FM88] You-Chin Fuh und Prateek Mishra: *Type Inference with Subtypes.* In: Ganzinger, Harald [Gan88], Seiten 94–114. (Zitiert auf den Seiten 108, 155, 162, 176 und 194.)

[FM89] You-Chin Fuh und Prateek Mishra: *Polymorphic Subtype Inference: Closing the Theory-Practice Gap.* In: J. Diaz und F. Orejas (Herausgeber): *TAPSOFT'89 — Proceedings of the International Joint Conference on*

Theory and Practice of Software Development, Band 352 der Reihe *Lecture Notes in Computer Science*, Seiten 167–183, Barcelona, März 1989. Springer-Verlag. (Zitiert auf den Seiten 189, 190 und 193.)

[FPC89] *The Fourth International Conference on Functional Programming Languages and Computer Architecture*, Imperial College, London, September 1989. (Zitiert auf den Seiten 223 und 226.)

[FPC91] *Functional Programming Languages and Computer Architecture, 5th ACM Conference*, Band 523 der Reihe *Lecture Notes in Computer Science*, Cambridge, Massachusetts, August 1991. Springer-Verlag. (Zitiert auf den Seiten 223 und 226.)

[FSVY91] T. Frühwirth, E. Shapiro, M. Y. Varai und E. Yardeni: *Logic Programs as Types for Logic Programs.* In: Albert R. Meyer (Herausgeber): *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, Seiten 300–309, Amsterdam, The Netherlands, Juli 1991. IEEE Computer Society Press. (Zitiert auf Seite 147.)

[Gan88] Harald Ganzinger (Herausgeber): *ESOP'88, 2nd European Symposium on Programming*, Band 300 der Reihe *Lecture Notes in Computer Science*, Nancy, France, März 1988. Springer-Verlag. (Zitiert auf den Seiten 217 und 221.)

[GG92] Benjamin Goldberg und Michael Gloger: *Polymorphic Type Reconstruction for Garbage Collection Without Tags.* In: *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming*, Seiten 53–65, San Francisco, CA, Juni 1992. ACM Press. (Zitiert auf Seite 95.)

[GJ79] Miachael A. Garey und David S. Johnson: *Computers and Intractability.* W. H. Freeman and Company, San Francisco, 1979. (Zitiert auf Seite 177.)

[GJSB00] James Gosling, Bill Joy, Guy Steele und Gilad Bracha: *The Java Language Specification, Second Edition.* Addison Wesley, 2000. (Zitiert auf Seite 35.)

[GKT90] Michael Gloger, Stefan Kaes und Christoph Thies: *Entwicklung funktionaler Programme in der* SAMPλE*-Programmierumgebung.* Technischer Bericht PI-R3/90, TH Darmstadt, Praktische Informatik, D-6100 Darmstadt, Juni 1990. (Zitiert auf den Seiten 67, 103 und 104.)

[Glo92] Michael Gloger: *Implementierung funktionaler Programmiersprachen. Codegenerierung, Test- und Laufzeitsysteme für Sprachen mit verzögerter Auswertung.* Informatik. Deutscher Universitäts Verlag, Wiesbaden, 1992. (Zitiert auf Seite 102.)

[Gol91] Benjamin Goldberg: *Tag-Free Garbage Collection for Strongly Typed Programming Languages.* In: *POPL* [POP91], Seiten 165–177. (Zitiert auf Seite 94.)

[GR80] H. Ganzinger und K. Ripken: *Operator identification in ADA: formal specification, complexity, and concrete implementation.* ACM SIGPLAN Notices, 15(2):30–42, Februar 1980. (Zitiert auf Seite 36.)

[Har86] Robert Harper: *Introduction to Standard ML.* Technischer Bericht ECS-LFCS-86-14, University of Edinburgh, November 1986. (Zitiert auf Seite 38.)

[Hen96] Fritz Henglein: *Syntactic Properties of Polymorphic Subtyping.* TOPPS Technical Report (D-report series) D-293, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, Mai 1996. (Zitiert auf den Seiten 145, 194 und 213.)

[HHN01] Michael Hanus, Frank Huch und Philipp Niederau: *ObjectCurry: An Object-Oriented Extension of the Declarative Multi-Paradigm Language Curry.* Lecture Notes in Computer Science, 2011:89–106, 2001. (Zitiert auf Seite 144.)

[HJW+91] Paul Hudak, Simon Peyton Jones, Philip Wadler et al.: *Report on the Functional Programming Language Haskell.* Technischer Bericht, Yale University, August 1991. (Zitiert auf den Seiten 88 und 97.)

[HM95] My Hoang und John C. Mitchell: *Lower bounds on type inference with subtypes.* In: ACM (Herausgeber): *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: San Francisco, California, January 22–25, 1995*, Seiten 176–185, New York, NY, USA, 1995. ACM Press. (Zitiert auf Seite 195.)

[HR97] Fritz Henglein und Jakob Rehof: *The Complexity of Subtype Entailment for Simple Types.* In: *Proceedings, Twelth Annual IEEE Symposium on Logic in Computer Science*, Seiten 352–361, Warsaw, Poland, 29 Juni–2 Juli 1997. IEEE Computer Society Press. (Zitiert auf Seite 194.)

[HS83] M. Hunkel und H. Schmidt: *Ein System zur Bezeichneridentifikation und dessen Integration in ein strukturorientiertes Ediersystem.* Diplomarbeit, TH Darmstadt, Fachbereich Informatik, 1983. (Zitiert auf Seite 74.)

[Hud89] Paul Hudak: *Conception, Evolution, and Application of Functional Programming Languages.* ACM Computing Surveys, 21(3):359–411, September 1989. Yale. (Zitiert auf Seite 151.)

[Hug87] John Hughes: *Backward Analysis of Functional Programs.* Departmental Research Report CSC/87/R3, University of Glasgow, März 1987. (Zitiert auf Seite 104.)

[Jäg87] Michael Jäger: *SAMPLE – Spezifikations- und Programmiersprache.* Technischer Bericht PI-R1/87, TH Darmstadt, Praktische Informatik, 1987. (Zitiert auf Seite 102.)

[Jäg88a] Michael Jäger: *The SAMPLE Environment.* In: Gregor Snelting (Herausgeber): *Sprachspezifische Programmierumgebungen*, Darmstadt, April 1988. Gesellschaft für Informatik, Fachgruppe Implementierung von Programmiersprachen. (Zitiert auf Seite 103.)

[Jäg88b] Michael Jäger: *SAMPLE Sprachbeschreibung, Version 2.* Technischer Bericht PI-R7/88, TH Darmstadt, Praktische Informatik, 1988. (Zitiert auf Seite 102.)

[JGK88] Michael Jäger, Michael Gloger und Stefan Kaes: *SAMPLE — A Functional Language*. In: Robin E. Bloomfield, Lynn S. Marshall und Roger B. Jones (Herausgeber): *Proceedings VDM'88, VDM — The Way Ahead*, Band 328 der Reihe *Lecture Notes in Computer Science*, Seiten 202–217, Dublin, September 1988. Springer-Verlag. (Zitiert auf Seite 102.)

[Jon92] Mark P. Jones: *A Theory of Qualified Types*. In: Bernd Krieg-Bruckner (Herausgeber): *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, Band 582 der Reihe *Lecture Notes in Computer Science*, Seiten 287–306. Springer-Verlag, New York, NY, 1992. (Zitiert auf Seite 144.)

[Jon93] Mark P. Jones: *A system of constructor classes: overloading and implicit higher-order polymorphism*. In: *FPCA '93: Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, Seiten 52–61, New York, N.Y., 1993. ACM Press. (Zitiert auf Seite 67.)

[Jon95] Mark P. Jones: *Simplifying and Improving Qualified Types*. In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, Seiten 160–169, La Jolla, California, Juni 25–28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press. (Zitiert auf den Seiten 144 und 145.)

[JW78] K. Jensen und N. Wirth: *Pascal User Manual and Report*. Springer-Verlag, 2. Auflage, 1978. (Zitiert auf Seite 35.)

[Kae88] Stefan Kaes: *Parametric Overloading in Polymorphic Programming Languages*. In: Ganzinger, Harald [Gan88], Seiten 131–144. (Zitiert auf den Seiten 61 und 77.)

[Kae92] Stefan Kaes: *Type Inference in the Presence of Overloading, Subtyping and Recursive Types*. In: *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, San Francisco, Juni 1992. (Zitiert auf den Seiten 129, 144 und 194.)

[KG89] Stefan Kaes und Michael Gloger: SAMPλE – *User Manual and Report*. TH Darmstadt, Praktische Informatik, Darmstadt, August 1989. Draft. (Zitiert auf Seite 102.)

[KM89] Paris C. Kanellakis und John. C. Mitchell: *Polymorhpic unfication and ML typing*. In: *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, Seiten 105–115, Januar 1989. (Zitiert auf den Seiten 18 und 29.)

[KR88] Brian W. Kernighan und Dennis M. Ritchie: *The C Programming Language, ANSI C*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 2. Auflage, 1988. (Zitiert auf Seite 35.)

[Lan66] Peter J. Landin: *The next 700 programming languages*. Communications of the ACM, 9(3):157–166, März 1966. (Zitiert auf Seite 73.)

[Let83] Thomas Letschert: *Type Inference in the Presence of Overloading and Coercions*. In: *Tagung der GI-Fachgruppe Compiler Construction*, Seiten 1–2, 1983. (Zitiert auf Seite 18.)

[Let86] Thomas Letschert: *Typinferenzsysteme*. Doktorarbeit, TH Darmstadt, Fachbereich Informatik, 1986. (Zitiert auf den Seiten 14, 20, 176 und 179.)

[Mai90] Harry G. Mairson: *Deciding ML typability is complete for deterministic exponential time*. In: ACM (Herausgeber): *POPL '90. Proceedings of the seventeenth annual ACM symposium on Principles of programming languages, January 17–19, 1990, San Francisco, CA*, Seiten 382–401, New York, NY, USA, 1990. ACM Press. (Zitiert auf Seite 29.)

[Meh84] Kurt Mehlhorn: *Data Structures and Algorithms, Vol. 2: Graph Algorithms and NP-Completeness*, Band 2 der Reihe *EATCS Monographs in Computer Science*. Springer-Verlag, 1984. (Zitiert auf Seite 182.)

[Mil78] Robin Milner: *A Theory of Type Polymorphism in Programming*. Journal of Computer and System Sciences, 17(3):348–375, Dezember 1978. (Zitiert auf den Seiten 1 und 17.)

[Mil84] Robin Milner: *A Proposal for Standard ML.* In: *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Seiten 184–197. ACM, August 1984. Invited paper. (Zitiert auf den Seiten 1 und 38.)

[Mit84] John C. Mitchell: *Coercion and Type Inference.* In: *POPL* [POP84], Seiten 175–185. (Zitiert auf den Seiten 108, 155, 162 und 189.)

[MM82] Alberto Martelli und Ugo Montanari: *An Efficient Unification Algorithm.* ACM Transactions on Programming Languages and Systems, 4(2):258–282, 1982. (Zitiert auf den Seiten 14 und 151.)

[Mog89] Eugenio Moggi: *Computational Lambda-Calculus and Monads.* In: *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989*, Seiten 14–23. IEEE Computer Society Press, Washington, DC, 1989. (Zitiert auf Seite 67.)

[MPS84] David B. MacQueen, Gordon Plotkin und Ravi Sethi: *An Ideal Model for Recursive Types.* In: *POPL* [POP84], Seiten 165–174. (Zitiert auf den Seiten 32 und 203.)

[MS82] D. B. MacQueen und Ravi Sethi: *A Semantic Model of Types for Applicative Languages.* In: *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, Seiten 243–252. ACM, ACM, August 1982. (Zitiert auf Seite 32.)

[Nip] Tobias Nipkow: *Equational Reasoning.* Course Notes. (Zitiert auf den Seiten 14 und 151.)

[NP93] T. Nipkow und C. Prehofer: *Type Checking Type Classes.* In: *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Seiten 409–418, New York, NY, 1993. ACM. (Zitiert auf Seite 93.)

[NS91] Tobias Nipkow und Gregor Snelting: *Type Classes and Overloading Resolution via Order-Sorted Unification.* In: *FPCA* [FPC91], Seiten 1–14. (Zitiert auf den Seiten 57, 73, 97 und 106.)

[Oho89] Atsushi Ohori: *A simple semantics for ML polymorphism.* In: *FPCA* [FPC89], Seiten 281–292. (Zitiert auf Seite 18.)

[Oph91] Dagmar Ophardt: *Benutzerdefinierbare Überladungen in* SAMPλE*, Konzepte und Implementierung.* Diplomarbeit, TH Darmstadt, Praktische Informatik, 1991. (Zitiert auf Seite 102.)

[OWW95] Martin Odersky, Philip Wadler und Martin Wehr: *A Second Look at Overloading.* In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, Seiten 135–146, La Jolla, California, Juni 25–28, 1995. ACM SIGPLAN/-SIGARCH and IFIP WG2.8, ACM Press. (Zitiert auf Seite 77.)

[POP84] *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, Januar 1984. (Zitiert auf Seite 222.)

[POP91] *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, Januar 1991. (Zitiert auf den Seiten 215 und 218.)

[PW78] M. Paterson und M. Wegman: *Linear Unification.* Journal of Computer and System Sciences, 16:158–167, 1978. (Zitiert auf Seite 14.)

[Reh97] Jakob Rehof: *Minimal typings in atomic subtyping.* In: ACM (Herausgeber): *Conference record of POPL '97, the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Paris, France, 15–17 January 1997*, Seiten 278–291, New York, NY, USA, 1997. ACM Press. (Zitiert auf den Seiten 193 und 195.)

[Rey85] John C. Reynolds: *Three Approaches to Type Structure.* In: *Mathematical Foundations of Software Development - Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, Band 185 der Reihe *Lecture Notes in Computer Science*, Seiten 97–138, Berlin, März 1985. Springer-Verlag. (Zitiert auf den Seiten 179 und 194.)

[Rob65] J. A. Robinson: *A machine-oriented logic based on the resolution principle.* Journal of the ACM, 12(1):23–41, Januar 1965. (Zitiert auf Seite 13.)

[SA93] Zhong Shao und Andrew W. Appel: *Smartest Recompilation.* In: *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Seiten 439–450, Charleston, South Carolina, Januar 10–13, 1993. ACM Press. (Zitiert auf Seite 144.)

[SB89] Gregor Snelting und Rolf Bahlke: *Why Theory-Based Environments are Better.* In: *Proceedings International Conference on System Development Environments & Factories*, Berlin, Mai 1989. North-Holland. (Zitiert auf Seite 3.)

[Sch85] David A. Schmidt: *Detecting Global Variables in Denotational Specifications.* ACM Transactions on Programming Languages and Systems, 7(2):299–310, April 1985. (Zitiert auf Seite 84.)

[Sch86] David A. Schmidt: *Denotational Semantics - A Methodology for Language Development.* Allyn and Bacon, 1986. (Zitiert auf Seite 197.)

[Sei94] Helmut Seidl: *Haskell overloading is DEXPTIME-complete.* Information Processing Letters, 52(2):57–60, Oktober 1994. (Zitiert auf Seite 50.)

[Sie89] Jörg H. Siekmann: *Unification Theory.* Journal of Symbolic Computation, 7:207–273, Februar 1989. (Zitiert auf Seite 131.)

[Sim89] K. Simon: *Finding a Minimal Transitive Reduction in a Strongly Connected Digraph within Linear Time.* Technical Report 1989TR-103, Swiss Federal Institute of Technology, Zurich, April, 1989. (Zitiert auf Seite 184.)

[Smi94] Geoffrey S. Smith: *Principal type schemes for functional programs with overloading and subtyping.* Science of Computer Programming, 23(2–3):197–226, Dezember 1994. Selected papers of the Colloquium on Formal Approaches of Software Engineering (Orsay, 1993). (Zitiert auf den Seiten 50, 108, 195 und 196.)

[Sne91] Gregor Snelting: *The calculus of context relations.* Acta Informatica, 28(5):411–445, Mai 1991. (Zitiert auf Seite 3.)

[SNGM89] G. Smolka, W. Nutt, J. A. Goguen und J. Meseguer: *Order-Sorted Equational Computation.* In: H. Aït-Kaci und M. Nivat (Herausgeber): *Resolution of Equations in Algebraic Structures, Volume 2: Rewriting Techniques*, Seiten 297–367. Academic Press, 1989. (Zitiert auf den Seiten 58 und 59.)

[SS85] Manfred Schmidt-Schauss: *A Many-Sorted Calculus with Polymorphic Functions Based on Resolution and Paramodulation.* In: Aravind Joshi (Herausgeber): *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, Seiten 1162–1168, Los Angeles, CA, August 1985. Morgan Kaufmann. (Zitiert auf Seite 59.)

[Sto77] Joseph E. Stoy: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977. (Zitiert auf den Seiten 31 und 197.)

[Str67] Christopher Strachey: *Fundamental Concepts of Programming Languages.* In: *International Summer School in Computer Programming*, Kopenhagen, 1967. (Zitiert auf Seite 3.)

[Tur85] David A. Turner: *Miranda: A non-strict Functional Language with Polymorphic Types.* In: Jean-Pierre Jouannaud (Herausgeber): *Functional Programming Languages and Computer Architecture*, Band 201 der Reihe *Lecture Notes in Computer Science*, Seiten 1–16, Nancy, September 1985. Springer-Verlag. (Zitiert auf Seite 1.)

[TW93] Jerzy Tiuryn und Mitchell Wand: *Type Reconstruction with Recursive Types and Atomic Subtyping.* In: Marie-Claude Gaudel und Jean-Pierre Jouannaud (Herausgeber): *TAPSOFT '93: Theory and Practice of Software Development, 4th International Joint Conference CAAP/FASE*, LNCS 668, Seiten 686–701, Orsay, France, April 13–17, 1993. Springer-Verlag. (Zitiert auf Seite 209.)

[Vol94] D. M. Volpano: *Haskell-style Overloading is NP-hard.* In: *Proceedings: 5th International Conference on Computer Languages*, Seiten 88–94. IEEE Computer Society Press, 1994. (Zitiert auf Seite 50.)

[Vol96] Dennis M. Volpano: *Lower bounds on type checking overloading.* Information Processing Letters, 57(1):9–13, Januar 1996. (Zitiert auf Seite 50.)

[VS91] Dennis Volpano und Geoffrey S. Smith: *On the Complexity of ML Typability with Overloading.* In: *FPCA* [FPC91], Seiten 15–28. (Zitiert auf Seite 50.)

[Wad90] Philip Wadler: *Comprehending Monads.* In: *1990 ACM Conference on Lisp and Functional Programming*, Seiten 61–78. ACM, Juni 1990. (Zitiert auf Seite 67.)

[Wad92] P. Wadler: *The essence of functional programming.* Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, Seiten 1–14, Januar 1992. (Zitiert auf Seite 67.)

[Wan87] Mitchell Wand: *A Simple Algorithm and Proof for Type Inference.* Fundamenta Infomaticae, 10:115–122, 1987. (Zitiert auf Seite 130.)

[WB89] P. Wadler und S. Blott: *How to make* ad-hoc *polymorphism less* ad hoc. In: ACM (Herausgeber): *POPL '89. Proceedings of the sixteenth annual ACM symposium on Principles of programming languages, January 11–13, 1989, Austin, TX*, Seiten 60–76, New York, NY, USA, 1989. ACM Press. (Zitiert auf den Seiten 88, 97 und 106.)

[Wir83] Niklaus Wirth: *Programming in Modula-2.* Springer-Verlag, Berlin, Heidelberg, London, 2. Auflage, 1983. (Zitiert auf Seite 35.)

[WO89] Mitchell Wand und Patrick O'Keefe: *On the Complexity of Type Inference with Coercion.* In: *FPCA* [FPC89], Seiten 306–324. (Zitiert auf den Seiten 176 und 194.)

Anhang A

Notation

$\mathbb{N}$	die Natürlichen Zahlen
$\mathbb{N}_+$	die Natürlichen Zahlen ohne 0
$\mathbf{2}^M$	die Potenzmenge über M
M^*	Wortmonoid über M
ϵ	die leere Sequenz eines Wortmonoids
μ, ν	Variablen über einem Wortmonoid
$\mu{\cdot}\nu$	Konkatenation in einem Wortmonoid
x, y, z	Variablen in Lambdatermen
α, β, γ	Typvariablen
$\overline{x_n}$	$x_1, \ldots, x_n$
a, b, c	atomare Typausdrücke (Variablen oder Basistypen)
τ	Typausdrücke
σ	Typschemata
Γ	Typzuweisungen der Form $[x_1{:}\tau_1, \ldots, x_n{:}\tau_n]$ bzw. $[x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n]$
t/p	Subterm von t an Position p
$f : A \to B$	(partielle) Abbildung von A nach B
$dom(f)$	Urbildbereich der partiellen Abbildung f
$rng(f)$	Bildbereich der partiellen Abbildung f
$f_{\mid M}$	Einschränkung einer Abbildung auf M
$\mid M \mid$	Kardinalität von M wobei $\mid M \mid \in \mathbb{N} \cup \{\infty\}$
R^*	Transitive Hülle einer Relation R

Lebenslauf

14. Oktober 1959	geboren in Frankfurt am Main
1966–1978	Schulausbildung
Juni 1978	Abitur am Helmholtz Gymnasium in Frankfurt
1978–1979	Wehrdienst
1979–1985	Studium der Informatik mit Nebenfach Mathematik an der TH Darmstadt
06/1985	Diplom in Informatik
06/1985–06/1992	Wissenschaftlicher Mitarbeiter am Fachbereich Informatik der TH Darmstadt, Fachgebiet Praktische Informatik
01/1992–01/2001	Dresdner Bank Frankfurt, zuletzt Leiter Competence Center DAP
02/2001–09/2001	portax.com GmbH München, Leiter Softwareentwicklung
seit 09/2001	freiberufliche Tätigkeit, u.a. für Siemens AG Österreich